The Receptive Distributed π-Calculus

Roberto M. Amadio, Gérard Boudol, Cédric Lhoussaine

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Receptive Distributed π-Calculus

Roberto Amadio — Gérard Boudol — Cédric Lhoussaine

N° 4080

Novembre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4080--FR+ENG

INRIA
SOPHIA ANTIPOLIS

The Receptive Distributed π-Calculus*

Roberto Amadio† , Gérard Boudol , Cédric Lhoussaine‡

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n° 4080 — Novembre 2000 — 39 pages

Abstract: In this paper we study an asynchronous distributed π-calculus, with constructs for localities and migration. We show that a simple static analysis ensures the receptiveness of channel names, which, together with a simple type system, guarantees a local deadlock-freedom property, that we call message deliverability. This property states that any migrating message will find an appropriate receiver at its destination locality. We argue that this distributed, receptive calculus is still expressive enough, by giving a series of examples illustrating the "receptive style" of programming we have. Finally we show that our calculus contains the π_1-calculus, up to weak asynchronous bisimulation.

Key-words: concurrency, asynchrony, distributed systems, mobility, π-calculus, types

* Work partially supported by the RNRT project MARVEL. This is a revised and expanded version of a paper appeared in the proceedings of the FST&TCS'99 Conference, Lecture Notes in Comp. Sci. 1738.
† CMi, Université de Provence
‡ CMi, Université de Provence

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Le π-calcul réparti réceptif

Résumé : Dans cet article nous étudions un π-calcul asynchrone réparti, muni de constructions de localités et de migration. Nous montrons qu'on peut garantir la réceptivité des canaux de communication grâce à une analyse statique très simple, et que ceci, combiné avec un sytème de types, garantit l'absence de blocage local. Nous appelons cette propriété la "livrabilité" des messages; elle exprime que tout message migrant trouvera un récepteur disponible dans sa localité de destination. Nous montrons que ce π-calcul réparti et réceptif reste très expressif, par une série d'exemples illustrant le style de programmation obtenu. Nous prouvons enfin que notre calcul contient le π_1-calcul, modulo la bisimulation asynchrone faible, ce qui établit son pouvoir d'expression, du moins en ce qui concerne le fragment sans migration.

Mots-clés : parallélisme, asynchronie, systèmes répartis, mobilité, π-calcul, types

1. Introduction

In this paper we study a sub-calculus of Hennessy and Riely's *distributed π-calculus* Dπ [11]. This is a calculus based on the polyadic π-calculus, involving explicit notions of *locality* and *migration*, that is code movement from a location to another. In this model communication is purely local, so that messages to remote resources (i.e. receivers, or input processes, in π-calculus terminology) must be explicitly routed. In such a model – as opposed to others like the JOIN calculus [9, 10] or the π_{1l}-calculus [2] where messages transparently go through the network to reach their (unique) destination – the problem arises of how to avoid local deadlocks. More precisely, we would like to avoid the situation where a message migrates to, or stays in a locality where no receiver will ever be available.

To solve this problem, we must ensure that a resource will be available when requested. An obvious way to achieve this is to enforce *receptiveness* – following Sangiorgi's terminology [16] – of any (private) channel name, that is the property that at any time a process is able of offering an input on that channel name. However, the kind of receptiveness we are looking for is neither the "uniform", nor the "linear" one described in [16]. Indeed, requiring each (private) name to be either uniform or linear receptive would result in a dramatic loss of expressive power. Then we are seeking for a formalization of something like the naming of a persistent "object", or, perhaps more appropriately, an "actor", which may encapsulate a state. Moreover, we also wish to ensure the desirable property of uniqueness of receivers, hence of "objects", that has been advocated by Fournet and Gonthier in [9] as a requirement for a language to be implementable in a distributed manner.

It is actually not very difficult to design a simple inference system for statically checking processes, in such a way that private names are (uniquely) receptive inside their scope: we define inductively a relation $I \Vdash P$, where I is a set of names, meaning that P is receptive with interface I. That is, I is the set of names on which P persistently provides a receiver. This relation, which refines the one of π_1, is given by means of a few rules. For instance an input cannot be nested inside another, unless they are inputs on the same name. This "well-formedness" predicate is not enough to entail the local deadlock-freedom property however, since we not only have channel names, but also location names, and other names, called keys, that may be compared for equality but cannot be used as communication channels; these are crucially needed to retain the expressive power, as we shall see. In particular, the restriction operation $(\nu u)P$ of the π-calculus must be refined, because we do not demand a location name or a key to be receptive, since there is no notion of a receiver for such a name, and still we want such a name to be sometimes restricted in scope. Types seem to be the obvious solution for this particular problem.

The type system we consider in this paper appears to be a simplified version of the simple type system of [11]: we use location types, that record the names and types of channels that may be used to communicate inside a locality, and we use "located types" for channels that are sent together with their location, instead of the more expressive existential types of [11]. We show the usual "subject reduction property", which holds for the system for checking receptiveness too. Then we are able to prove that typed receptive processes do not run into the kind of local deadlocks we wished to avoid. We call this property *message deliverability*, since it states that any message will find an appropriate receiver in its destination locality.

The issue of avoiding deadlocks by statically inferring some properties has already been addressed. For instance, in [5], a type system intended to prevent deadlocks is proposed for the blue calculus, where the types are formulae of Hennessy-Milner Logic with recursion. Kobayashi also studied in [12, 20] means to avoid deadlocks in the π-calculus. His approach is quite different from ours, however: he uses a sophisticated type system where types involve time tags and where typing contexts record an ordering on the usage of names, whereas we only use a very simple information – the set of receiver's names. Since we regard receivers as passive entities, providing resources for names, we are not seeking to avoid the situation where no message is sent to a receiver – in this situation the resource provider should be garbage collected.

The question arises whether the nice properties of our receptive distributed π-calculus are obtained at the expense of its expressivity – we immediately notice that expressing functional computations, as in [21, 18] for instance, translates easily into the (uniform) receptive style. Indeed, receptiveness is both a strong safety property and a quite heavy constraint. We therefore illustrate, by a series of examples, culminating in various kinds of migrating reference cells, the "receptive style" of programming we need to adopt; this tends to be quite similar to the object oriented style, where a receiver (in the π-calculus sense) is a named persistent "object", reacting to messages according to some protocol, where the keys play the rôle of "selectors". We also establish a result that proves that expressive power is not lost: we show that the π_1-calculus of Amadio

$\tau,\ \sigma \ldots$	::=	$\zeta \mid \psi \mid \gamma^{@}$	*types*
ζ	::=	$val \mid \gamma$	*values and channel types*
$\gamma,\ \delta \ldots$	::=	$Ch(\tau_1, \ldots, \tau_n)$	*channel types*
$\psi,\ \phi \ldots$	::=	$\{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$	*location types*

Figure 1: Types

[2], in which the JOIN calculus can be encoded [3], may be translated into our receptive π_1-calculus, in a way which is fully abstract with respect to weak asynchronous bisimulation.

2. Distributed Processes.

In this section we introduce our calculus of distributed processes, which may be seen as a subcalculus of the distributed π-calculus of Hennessy and Riely [11]. This is basically the asynchronous, polyadic π-calculus, with some primitives for spatial distribution of processes and migration, organized as a two-level model: besides the usual notion of a *process* – called a "thread" in [11] –, performing inputs and outputs on communication channels, we consider *networks of located processes* – called "systems" in [11], and "configurations" in [2].

In order to state the syntax, we assume, as usual, that a denumerable set $\mathcal{N}$ of (simple) *names*, ranged over by $a,\ b,\ c, \ldots$, and also sometimes $x,\ y,\ z, \ldots$, is given. We will informally distinguish a subset of names, that we denote $\ell, \ell', \ldots$, which are supposed to name *localities* – formally, there will be different kinds of types for names. Then the processes may also send and receive *compound names*, which are pairs of simple names that we write $a@\ell$, meaning "the (channel) name a used at location ℓ". We use $u,\ v,\ w, \ldots$ to denote simple or compound names, and a (possibly empty) vector of such names is written $\vec{u}$. We shall often use the operation $\_@\ell$ on (compound) names, defined by

$$u@\ell = \begin{cases} a@\ell & \text{if } u = a \\ u & \text{if } u = a@\ell' \end{cases}$$

This operation is extended to sets of names in the obvious way (i.e. $N@L = \{\, u@\ell \mid u \in N \ \&\ \ell \in L \,\}$).

As we suggested, and as it is normally done in the polyadic π-calculus, the names are used according to a typing discipline, and the processes are checked to obey this discipline. The types we use generalize the usual sorts of the π-calculus, and in particular channel sorts will be types. We also have to assign types to location names, and to compound names. The idea here is that a location type should record the names and types of channels on which communication is possible inside a locality. Therefore, a location type is just a typing context, or a record type of a certain kind, as in [11], while the type of a located channel is a "located channel type" $\gamma^{@}$, which is simpler than an existential type, as used in [11]. We also introduce a type val, for names that may be tested for equality, but are not used as communication channels or location names; we allow the latter to be compared too. In a richer language we would refine this val type, distinguishing in particular the types of values that can be tested for equality. The types (for names) are given in the Figure 1. In the notation $\{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$ it is understood that the names $a_1, \ldots, a_n$ are all distinct, and that their order does not matter. Then a location type ψ is really a mapping from a finite set of simple names into the set of channel types. We shall sometimes denote by loc the empty location type. We also denote by $\psi \sqcup \phi$ the union of ψ and ϕ, which is only defined if ψ and ϕ assign the same types to the names they share.

In the syntax, where we use explicit recursive definitions instead of replication (see [15, 21]), we also need to use process identifiers – or, rather, parametric process identifiers –, denoted $A, B, \ldots$ and belonging to a denumerable set $\mathcal{P}$, disjoint from $\mathcal{N}$. These are parametric on names, and we shall write a recursive call as $A(\vec{u})$. The grammar for terms is given in the Figure 2. We denote by $U,\ V \ldots$ a term of any kind – process, parametric process or network. There are some requirements for terms to be legal, that we leave informal, which are as follows:

(i) in an input $a(\vec{u}).P$, all the names occurring in $\vec{u}$ are assumed to be distinct, and different from a, and similarly in $(\mathsf{rec}\, A(\vec{u}).P)$.
(ii) we make the standard assumption that recursion is *guarded*, that is in $(\mathsf{rec}\, A(\vec{u}).P)(\vec{v})$ all recursive calls to A in P occur under an input guard.

$u, v \ldots$	::=		*names*
		a	*simple*
	|	$a@\ell$	*compound*
w	::=	$u \mid a : val$	
$P, Q, R \ldots$	::=		*processes*
		$\mathbf{0}$	*inaction*
	|	$\overline{a}(u_1, \ldots, u_n)$	*message*
	|	$a(u_1, \ldots, u_n).P$	*input on channel a*
	|	$(P \mid Q)$	*parallel composition*
	|	$[a = b]P, Q$	*conditional branching*
	|	$(\nu w)P$	*scoping (restriction)*
	|	$\mathsf{go}\,\ell.P$	*migration*
	|	$T(u_1, \ldots, u_n)$	*parametric process instanciation*
T	::=		*parametric processes*
		A	*identifier*
	|	$(\mathsf{rec}\, A(u_1, \ldots, u_n).P)$	*recursive parametric process*
S	::=		*networks*
		$\mathbf{0}$	*empty network*
	|	$[\ell :: P]$	*located process*
	|	$(S \mid S')$	*parallel composition*
	|	$(\nu w)S \mid (\nu \ell : \psi)S$	*scoping (restriction)*

Figure 2: Syntax

(iii) in $(\nu \ell : \psi)S$ we assume that the name ℓ does not occur in the type ψ.

A process $a(\vec{u}).P$ is also called a *receiver* (on channel a), but we shall often use the names "server" or "resource" (of name a) for such an agent, a terminology that will be justified later. For any w, we define its subject $\mathsf{subj}(w)$ as follows: $\mathsf{subj}(a) = a = \mathsf{subj}(a@\ell) = \mathsf{subj}(a : \tau)$. In $(\nu w)P$, the subject of w is the only name occurring in w which is bound. Apart from that, the notions of free and bound names are the usual ones; we denote by $\mathsf{fn}(U)$ and $\mathsf{bn}(U)$ the sets of free and bound names respectively occurring in the term U, and $\mathsf{nm}(U)$ denotes the union of these two sets (we shall reuse this notation, for instance for vectors of compound names). We shall say that U is *closed* if it does not contain any free process identifier (notice that a closed term may contain free names).

The need for compound names is explained by Hennessy and Riely in [11], although they use more sophisticated "dependent names"; we will see later examples showing how to use such names. As regards located processes, we use the notation $[\ell :: P]$ of the construction for explicit spatial distribution of processes introduced in [6]. This is denoted $\ell[\![P]\!]$ and $\{P\}\ell$ in [11] and [2], respectively. We could also have used the notation $\ell[P]$ of the Ambient calculus of Cardelli and Gordon [8].

As a running example for this paper, we shall use a simple formulation of the standard *remote procedure call* mechanism; the dual of this, that is *remote evaluation*, is trivially expressed in our language, by the primitive operation $\mathsf{go}\,\ell.P$. In this example, a client process Q, located at ℓ, wants to call a service named a, located at ℓ', with arguments $\vec{d}$, and waits for results, to be processed by a continuation agent P. Then the client creates a private return channel at its own location, and sends it, as the subject of a compound name indicating the return locality, together with the data $\vec{d}$. We write this as follows:

$$Q = (\nu r)(\,\mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell) \mid r(\vec{v}).P\,)$$

To describe the service – the remote procedure –, let us first recall the "replicated input", given by

$$a^*(\vec{u}).P \ =_{\rm def} \ \mathsf{rec}\, A(a).a(\vec{u}).(P \mid A(a))$$

where we simply write $\mathsf{rec}\, A(\vec{u}).P$ for $(\mathsf{rec}\, A(\vec{u}).P)(\vec{u})$. Now assume that the service R computes a function $f(\vec{d})$ of the arguments and sends back the result to the caller; using an enriched syntax, we can write this simply as

$$R = a^*(\vec{x}, y@z).\mathsf{go}\, z.\overline{y}(f(\vec{x}))$$

One sees in particular how the compound name $y@z$ is decomposed into a locality, where to go, and a channel on which to send the result. In this example the migrating agents are just messages. We shall see more elaborate examples later, but since migrating messages are very often used, we introduce a specific notation for them:

$$\overline{a}@\ell(\vec{u}) =_{\rm def} \mathsf{go}\, \ell.\overline{a}(\vec{u})$$

Then for instance the RPC example is $[\ell' :: R] \mid [\ell :: Q]$ where

$$\begin{array}{rcl} Q & = & (\nu r)(\, \overline{a}@\ell'(\vec{d}, r@\ell) \ \mid \ r(\vec{v}).P\,) \\ R & = & a^*(\vec{x}, y@z).\overline{y}@z(f(\vec{x})) \end{array}$$

We now describe formally how a system behaves, by means of a *reduction relation* $U \to U'$. The definition of this relation involves the notion of *substitution*. We use two kinds of substitutions, of names for names, and of parametric processes for identifiers. A name substitution is a mapping S from a finite subset $\mathsf{dom}(\mathsf{S})$ of $\mathcal{N}$ into $\mathcal{N}$. We denote by $\mathsf{im}(\mathsf{S})$ the image of S, that is the set $\{\, \mathsf{S}(a) \mid a \in \mathsf{dom}(\mathsf{S}) \,\}$. Applying the substitution to a term U is denoted $[\mathsf{S}]U$, but *this is only defined if no capture may arise*, that is if $\mathsf{im}(\mathsf{S}) \cap \mathsf{bn}(U) = \emptyset$. Moreover, whenever U is a network term, we require S to be *injective* for $[\mathsf{S}]U$ to be defined; the reason for this is that in the case of $U = (\nu\ell : \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\})S$ the names $a_1, \ldots, a_n$ are free, and possibly in $\mathsf{dom}(\mathsf{S})$, but must be kept distinct after substitution. The definitions and technical results regarding substitutions are collected in the Appendix. If S is the mapping

$$a_1 \mapsto b_1, \ldots, a_k \mapsto b_k$$

then $[\mathsf{S}]U$ is also denoted $[b_1/a_1, \ldots, b_k/a_k]U$, or $[\vec{b}/\vec{a}]U$. We also use the notation $[v/u]U$ where u and v are possibly compound names; in this case, if $u = x@y$ then x is supposed to be distinct from y, and v must be a compound name $a@b$, and the substitution really is $[a/x, b/y]U$, whereas if u is a simple name, then v must be a simple name too. Regarding the substitution of parametric processes for indentifiers, we shall only use substitutions for a single name in process terms, written $[T/A]P$. Again, this is only defined if no capture may arise, that is $\mathsf{fn}(T) \cap \mathsf{bn}(P) = \emptyset$.

As usual, terms are considered identical if they are equal up to renaming of bound names that is, if they are α-convertible (see the Appendix for this notion), but this is not the only case for equality we need to consider here: to define the reduction relation, we use the "chemical style" of Berry and Boudol [4], as popularized by Milner [15], that is using a *structural equivalence* relation. This is the least equivalence $\equiv$ containing the following axioms:

$(\mathbf{0} \mid U)$	$\equiv$	U		*neutrality*
$(U \mid (V \mid W))$	$\equiv$	$((U \mid V) \mid W)$		*associativity*
$(U \mid V)$	$\equiv$	$(V \mid U)$		*commutativity*
$((\nu w)U \mid V)$	$\equiv$	$(\nu w)(U \mid V)$	$\mathsf{subj}(w) \notin \mathsf{fn}(V)$	*scope migration* (1)
$[\ell :: P \mid Q]$	$\equiv$	$[\ell :: P] \mid [\ell :: Q]$		*routing*
$[\ell :: (\nu w)P]$	$\equiv$	$(\nu w@\ell)[\ell :: P]$	$\ell \neq \mathsf{subj}(w)$	*scope migration* (2)

where

$$w@\ell = \begin{cases} a@\ell & \text{if } w = a \\ w & \text{if } w = a@\ell' \text{ or } w = a : val \end{cases}$$

and satisfying the following rules

$$\frac{U =_\alpha V}{U \equiv V} \qquad \frac{U \equiv V \ , \ V \equiv W}{U \equiv W} \qquad \frac{U \equiv V}{\mathbf{E}[U] \equiv \mathbf{E}[V]}$$

$$
\begin{array}{rcll}
(\overline{a}(\vec{v}) \mid a(\vec{u}).P) & \to & [\vec{v}/\vec{u}]P & \textit{communication law} \\
[\ell' :: \mathsf{go}\,\ell.P] & \to & [\ell :: P] & \textit{law of movement} \\
(\mathsf{rec}\,A(\vec{u}).P)(\vec{v}) & \to & [(\mathsf{rec}\,A(\vec{u}).P)/A][\vec{v}/\vec{u}]P & \textit{unfolding} \\
[a = a]P, Q & \to & P & \textit{matching} \\
[a = b]P, Q & \to & Q \qquad a \neq b & \textit{mismatching}
\end{array}
$$

Figure 3: Reduction Laws

where $\mathbf{E}$ is any *evaluation context*, given by the following grammar:

$$\mathbf{E} ::= [] \mid (\mathbf{E} \mid U) \mid (\nu w)\mathbf{E} \mid [\ell :: \mathbf{E}]$$

Here $[]$ is "the hole", and filling the hole in a context $\mathbf{E}$ by a term U results, if this makes sense, in a term denoted $\mathbf{E}[U]$. The law of "routing" is mainly used, from left to right, to allow a migrating process to exit a locality, as in $[\ell :: \mathsf{go}\,\ell'.P \mid Q] \equiv [\ell :: \mathsf{go}\,\ell'.P] \mid [\ell :: Q]$ – we shall see that this evolves into $[\ell' :: P] \mid [\ell :: Q]$ –, and, from right to left, to allow a "packet" $[\ell' :: P]$ to join the destination locality. Up to this equivalence, any process P can be turned into a "canonical form", of the shape:

$$P \equiv (\nu \overrightarrow{w})(R_1 \mid \cdots \mid R_n)$$

where the R_i's, that we may call "molecules", are either messages $\overline{a}(\vec{v})$, or receivers $a(\vec{u}).Q$, or branching processes $[a = b]Q_0, Q_1$, or else recursive or migrating processes $T(\vec{u})$ and $\mathsf{go}\,\ell.Q$. Similarly, any network S can be turned into a canonical form:

$$S \equiv (\nu \cdots)([\ell_1 :: P_1] \mid \cdots \mid [\ell_k :: P_k])$$

where each P_1 is a parallel composition of molecules, and the ℓ_i's are pairwise distinct. For instance, in our RPC example, we have:

$$[\ell' :: R] \mid [\ell :: Q] \equiv (\nu r@\ell)\big([\ell' :: a^*(\vec{x}, y@z).\overline{y}@z(f(\vec{x}))] \mid [\ell :: \overline{a}@\ell'(\vec{d}, r@\ell) \mid r(\vec{v}).P]\big)$$

Now the axioms of reduction tell how the molecules react and interact. These axioms are given in Figure 3. Moreover, there are two rules, by which reduction is defined up to structural equivalence (hence in particular up to α-conversion), and may occur in any evaluation context:

$$\frac{V \equiv U \ , \ U \to U' \ , \ U' \equiv V'}{V \to V'} \qquad\qquad \frac{U \to U'}{\mathbf{E}[U] \to \mathbf{E}[U']}$$

Notice that in the communication law and unfolding we implictly assume that the substitution is defined. This means for instance, in the case of a communication, that the vectors $\vec{u}$ and $\vec{v}$ have the same length, and the same kind (simple or compound) of components. Moreover, the names sent in a message must not be bound in the receiver. However, there is no real restriction here, since we can always rename the bound names of the receiver to make the communication possible. We shall denote by $U \twoheadrightarrow V$ the reduction relation obtained by forbidding the use of the communication law in the definition of $\to$. One could prove that this notion of reduction is strongly normalizing, that is, there are only finite sequences of $\twoheadrightarrow$-reductions starting from any term, thanks to the assumption that recursion is guarded. One can see for instance, using the laws of structural equivalence, that

$$[\ell :: \mathsf{go}\,\ell'.P \mid Q] \mid [\ell' :: R] \twoheadrightarrow [\ell :: Q] \mid [\ell' :: P \mid R]$$

We could have taken this as the primitive law of movement, thus avoiding the use of the "routing" law ([1]). We should point out that *communication is only local*: typically, no reduction can arise from $[\ell :: \overline{a}(\vec{u})] \mid [\ell' :: a(\vec{v}).P]$ if $\ell \neq \ell'$, and one has to explicitly move and meet in order to communicate, like in $[\ell :: \mathsf{go}\,\ell'.\overline{a}(\vec{u})] \mid$

([1]) notice that the two formulations are not quite equivalent, since our simple law allows to create a locality. We do not use this feature though, and we could have introduced an explicit notion of a "packet", but this would slightly complicate the formal developments.

$[\ell' :: a(\vec{v}).P]$. In other words, unlike in [2, 10] where messages can transparently go through domains to reach the corresponding receiver, we have the "go and communicate" semantics of the Dπ-calculus, which is also the one of the Ambient calculus [8]. Coming back to our RPC example, we have:

$$\begin{array}{rcl}
[\ell' :: R] \mid [\ell :: Q] & \equiv & (\nu r@\ell)\big([\ell' :: R] \mid [\ell :: \mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell)] \mid [\ell :: r(\vec{v}).P] \big) \\
& \twoheadrightarrow & (\nu r@\ell)\big([\ell' :: R] \mid [\ell' :: \overline{a}(\vec{d}, r@\ell)] \mid [\ell :: r(\vec{v}).P] \big) \\
& \equiv & (\nu r@\ell)\big([\ell' :: R \mid \overline{a}(\vec{d}, r@\ell)] \mid [\ell :: r(\vec{v}).P] \big)
\end{array}$$

Since $R \twoheadrightarrow a(\vec{x}, y@z).(\mathsf{go}\, z.\overline{y}(f(\vec{x})) \mid R)$, we then have:

$$\begin{array}{rcl}
[\ell' :: R] \mid [\ell :: Q] & \stackrel{*}{\rightarrow} & (\nu r@\ell)\big([\ell' :: R \mid \mathsf{go}\,\ell.\overline{r}(f(\vec{d}))] \mid [\ell :: r(\vec{v}).P] \big) \\
& \equiv & (\nu r@\ell)\big([\ell' :: R] \mid [\ell' :: \mathsf{go}\,\ell.\overline{r}(f(\vec{d}))] \mid [\ell :: r(\vec{v}).P] \big) \\
& \twoheadrightarrow & (\nu r@\ell)\big([\ell' :: R] \mid [\ell :: \overline{r}(f(\vec{d}))] \mid [\ell :: r(\vec{v}).P] \big) \\
& \equiv & (\nu r@\ell)\big([\ell' :: R] \mid [\ell :: \overline{r}(f(\vec{d})) \mid r(\vec{v}).P] \big) \\
& \rightarrow & (\nu r@\ell)\big([\ell' :: R] \mid [\ell :: [f(\vec{d})/\vec{v}].P] \big)
\end{array}$$

Here we have assumed that the vectors $\vec{d}$ and $\vec{x}$ have the same length, and similarly for $f(\vec{d})$ and $\vec{v}$. This will be checked with the type system. This example demonstrates the use of compound names. However, in this example we could have used pairs of names r, ℓ (and y, z) instead of $r@\ell$ (resp. $y@z$). The difference will appear with the typing of processes: typically, a term like $a^*(\vec{x}, y@z).\overline{y}@z'(f(\vec{x}))$ will only be well-typed if $z' = z$.

3. A Simple Type System.

We have seen what the types are, and how they are used in the syntax: in fact only private values and locations need to be explicitly typed. Now we introduce a system for checking that a given term is acceptable with respect to some typing assumption regarding the names. As usual, typing should prevent some run-time errors, like arity mismatch between input and output on the same channel; Hennessy and Riely have proved in [11] that more sophisticated errors are avoided by their typing discipline. We do not repeat their result here, because we mainly use the typing information, in the proof of the "message deliverability" property, to distinguish names that are used as channels from names that are used as localities. Notice that, as with the ordinary π-calculus, distinguishing two kinds of names would not be enough, since names may be received, and then used as channels or localities.

The type system deals with sequents of the form $\Gamma \vdash_\ell P$, for checking that the process P, placed at the current location ℓ, conforms to the typing assumption Γ, and $\Gamma \vdash S$ and $\Gamma \vdash_\ell T : \gamma$, respectively for networks of located processes and for parametric processes. The system also uses an auxiliary system for typing names, whose sequents have the form $\Gamma \vdash_\ell u_1 : \tau_1, \ldots, u_n : \tau_n$. In these sequents the typing context Γ is a mapping from a finite subset $\mathsf{dom}(\Gamma)$ of $\mathcal{N} \cup \mathcal{P}$ into the set of types, subject to the following requirements:

(i) if $a \in \mathsf{dom}(\Gamma) \cap \mathcal{N}$ then $\Gamma(a)$ is either val or a locality type ψ.
(ii) if $A \in \mathsf{dom}(\Gamma) \cap \mathcal{P}$ then $\Gamma(A)$ is a channel type.
(iii) if $a \in \mathsf{dom}(\Gamma)$ then a does not occur in $\mathsf{im}(\Gamma)$, that is, a does not occur in any type assigned by Γ.

We will write a typing context as usual, that is as a list of typing assumptions:

$$\Gamma = \ldots,\ a : val\,, \ldots,\ \ell : \{c_1 : \gamma_1, \ldots, c_n : \gamma_n\}\,, \ldots,\ A : Ch(\vec{\tau})\,, \ldots$$

The requirement (iii) tells that, for instance, $a : val, \ell : \{a : Ch()\}$ is not a valid typing context. In the type system we make use of a partial operation of union Δ, Γ of typing contexts, defined as follows:

$$(\Delta, \Gamma)(x) = \begin{cases} \Delta(x) & \text{if } x \in \mathsf{dom}(\Delta) - \mathsf{dom}(\Gamma) \text{ or } \Delta(x) = \Gamma(x) \\ \phi \sqcup \psi & \text{if } \Delta(x) = \phi \ \&\ \Gamma(x) = \psi \\ \Gamma(x) & \text{if } x \in \mathsf{dom}(\Gamma) - \mathsf{dom}(\Delta) \end{cases}$$

There are several ways in which Γ, Δ may not be defined: for instance Γ could assign the type val to the name a, whereas $\Delta(a)$ is a location type, or some $a \in \mathsf{dom}(\Gamma)$ could occur in a type assigned by Δ, etc.

$$\frac{}{x : \tau \vdash_\ell x : \tau} \qquad \frac{}{\ell : \{a : \gamma\} \vdash_\ell a : \gamma} \qquad \frac{}{\ell : \{a : \gamma\} \vdash_{\ell'} a@\ell : \gamma^{@}}$$

$$\frac{\Gamma \vdash_\ell \overrightarrow{u : \tau} \quad , \quad \Delta \vdash_\ell \overrightarrow{v : \sigma}}{\Gamma , \Delta \vdash_\ell \overrightarrow{u : \tau}, \overrightarrow{v : \sigma}}$$

Figure 4: Type system for names

$$\frac{}{\vdash_\ell \mathbf{0}} \qquad \frac{\Gamma \vdash_\ell \overrightarrow{u : \tau}}{\ell : \{a : Ch(\vec{\tau})\} , \Gamma \vdash_\ell \overline{a}\vec{u}} \qquad \frac{\Gamma , \Delta \vdash_\ell P \quad , \quad \Gamma \vdash_\ell \overrightarrow{u : \tau}}{\ell : \{a : Ch(\vec{\tau})\} , \Delta \vdash_\ell a(\vec{u}).P}$$

$$\frac{\Gamma \vdash_\ell P \ , \ \Gamma \vdash_\ell Q}{\Gamma \vdash_\ell (P \mid Q)}$$

$$\frac{\Gamma \vdash_\ell P \ , \ \Gamma \vdash_\ell Q}{a, b : val \ , \Gamma \vdash_\ell [a = b]P, Q} \qquad \frac{\Gamma \vdash_\ell P \ , \ \Gamma \vdash_\ell Q}{a, b : loc \ , \Gamma \vdash_\ell [a = b]P, Q}$$

$$\frac{a : val \ , \Gamma \vdash_\ell P}{\Gamma \vdash_\ell (\nu a : val)P} \qquad \frac{\ell : \{a : \gamma\} \ , \Gamma \vdash_\ell P}{\Gamma \vdash_\ell (\nu a)P} \qquad \frac{\ell : \{a : \gamma\} \ , \Gamma \vdash_{\ell'} P}{\Gamma \vdash_{\ell'} (\nu a@\ell)P}$$

$$\frac{A : Ch(\vec{\tau}) \ , \Gamma \ , \Delta \vdash_\ell P \quad , \quad \Gamma \vdash_\ell \overrightarrow{u : \tau}}{\Delta \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P) : Ch(\vec{\tau})} \qquad \frac{\Delta \vdash_\ell T : Ch(\vec{\tau}) \quad , \quad \Gamma \vdash_\ell \overrightarrow{u : \tau}}{\Delta , \Gamma \vdash_\ell T(\vec{u})}$$

$$\frac{\Gamma \vdash_\ell P}{\Gamma \vdash_{\ell'} \mathsf{go}\, \ell.P} \qquad \frac{\Gamma \vdash_\ell P}{\Delta , \Gamma \vdash_\ell P}$$

Figure 5: Type system for processes

$$\frac{}{\vdash \mathbf{0}} \qquad \frac{\Gamma \vdash_\ell P}{\Gamma \vdash [\ell :: P]}$$

$$\frac{\Gamma \vdash S \ , \ \Gamma \vdash S'}{\Gamma \vdash (S \mid S')} \qquad \frac{\ell : \{a : \gamma\} \ , \Gamma \vdash S}{\Gamma \vdash (\nu a@\ell)S} \qquad \frac{a : \tau \ , \Gamma \vdash S}{\Gamma \vdash (\nu a : \tau)S} \qquad \frac{\Gamma \vdash S}{\Delta , \Gamma \vdash S}$$

Figure 6: Type system for networks

In Figure 4 the type system for names is presented, where x stands for any name of $\mathcal{N} \cup \mathcal{P}$. The meaning of $\Gamma \vdash_\ell u : \tau$ is that, given the type assumption Γ, u has type τ when used at location ℓ. Then for instance the axiom $\ell : \{a : \gamma\} \vdash_{\ell'} a@\ell : \gamma^{@}$ says that a compound name may be used at any location ℓ', with a located channel type, while a simple channel name may only be used at the location which holds it (with a simple channel type). We notice that the following holds:

REMARK 3.1. *If $\Gamma \vdash_\ell \overrightarrow{u : \tau}$ and $\Delta \vdash_{\ell'} \overrightarrow{u : \tau}$ then $\Gamma = \Delta$.*

In the typing rules for processes, collected in Figure 5, we use the following:

CONVENTION. *In the rules for the binding constructs, that is input, restriction and recursion, we implicitly have the usual condition that the bound names do not occur free in the resulting sequent.*

For instance, in the rule for the input construct, the names in $\vec{u}$ must not occur in Δ, $\{a, \ell\}$ and $\vec{\tau}$. Let us comment the rules of the system. The rules for the output and the input constructs are quite standard; let us just notice that the rule for output, combined with the weakening rule, involves a form of subtyping for

localities (*cf.* [11]): for instance, the judgement

$$\ell : \{b : \gamma_0, c : \gamma_1\}\ ,\ \ell' : \{a : Ch(loc)\} \vdash_{\ell'} \overline{a}(\ell)$$

is valid (recall that *loc* is $\{\}$), even though the type of ℓ given by the context is more generous than the one carried by the channel a. This form of subtyping is not allowed in input, since the received names cannot occur in the typing context. The only point to note about typing parallel composition is that the component must agree on the type of names they use, since they share the same context (one may have to use the weakening rule to ensure this). In the rules for conditional branching, we see that we can compare both names of value and location type, but not channel names. There are three cases for typing a restriction $(\nu w)P$ – the rules follow the same pattern as the ones for typing names. To type a migrating process $\mathsf{go}\,\ell.P$, one must be able to type P at locality ℓ, while the resulting current locality is immaterial. The terms of the ordinary asynchronous π-calculus with recursion are obviously terms of our distributed calculus, not involving any locality-based feature (compound names, migration). For typing these terms we use only the types given by

$$\kappa ::= val \ \mid\ Ch(\kappa_1, \ldots, \kappa_n)$$

Then we get the usual typing of these terms – ignoring recursion – (see [21] for instance) by transforming the judgements $\Delta \vdash P$ into $\Gamma \vdash_\ell P$ where ℓ is a fresh name, and

$$\Gamma = \{\, a : val \mid a : val \in \Delta \,\} \cup \{\ell : \{\, a : \kappa \mid \kappa \neq val \ \&\ a : \kappa \in \Delta \,\}\}$$

Now let us see some examples. The term $a(x@y).\overline{x}@y()$ is typable as follows – recall that $\overline{x}@y() = \mathsf{go}\,y.\overline{x}()$:

$$\dfrac{\dfrac{\dfrac{}{y : \{x : Ch()\} \vdash_y \overline{x}()}}{y : \{x : Ch()\} \vdash_\ell \mathsf{go}\,y.\overline{x}()} \qquad \dfrac{}{y : \{x : Ch()\} \vdash_\ell x@y : Ch()^{@}}}{\ell : \{a : Ch(Ch()^{@})\} \vdash_\ell a(x@y).\overline{x}@y()}$$

More generally, to type $a(x@y).P$ requires to type P, at some current location ℓ, in a context $y : \{x : \gamma\}\ , \Gamma$ where neither x nor y occur in Γ. This means that in P the channel x may only be used at location y. On the other hand, the term $a(x, y).\overline{x}@y()$ is not typable: to type the input one should use the valid sequent

$$y : \{x : Ch()\} \vdash_y x : Ch()\ ,\ y : \{x : Ch()\}$$

but this is not possible since the current location y would occur in the conclusion, contradicting our convention about bound names. Indeed, we want to keep a tight control on where the channels may be used, and allowing $a(x, y).\overline{x}@y()$ to be typable (say with an axiom $\ell : \{a : \gamma\} \vdash_{\ell'} a : \gamma$), it would be difficult not to accept also $a(x, y).\overline{x}@z()$, where a message may be sent at an unknown locality. In a similar vein, one can see that the term $a(y).\overline{b}@y()$ is typable as follows:

$$\dfrac{\dfrac{\dfrac{}{y : \{b : Ch()\} \vdash_y \overline{b}()}}{y : \{b : Ch()\} \vdash_\ell \overline{b}@y()} \qquad \dfrac{}{y : \{b : Ch()\} \vdash_\ell y : \{b : Ch()\}}}{\ell : \{a : Ch(\{b : Ch()\})\} \vdash_\ell a(y).\overline{b}@y()}$$

Still in the same vein of moving messages, we let the reader check that the following "forwarder" – similar to the "links" of [14]:

$$Fwd(a, b@\ell) = a^*(\vec{x}).\overline{b}@\ell(\vec{x})$$

is typable at any current locality ℓ' which holds the channel a, assuming that the x's are values, as follows:

$$\ell' : \{a : Ch(\overrightarrow{val})\}\ ,\ \ell : \{b : Ch(\overrightarrow{val})\} \vdash_{\ell'} Fwd(a, b@\ell)$$

There are terms of our calculus which are not typable in our system, but are accepted in the simple type system of Hennessy and Riely [11]: our located channel types $\gamma^{@}$ are particular cases of existential

types, namely $loc[\gamma]$, but the later are more expressive. For instance, we cannot type $a(x@y).\mathsf{go}\, y.(\overline{x}() \mid \overline{b}())$, because the subterm $\mathsf{go}\, y.(\overline{x}() \mid \overline{b}())$ can only be typed – at any current location – in a context $y : \{x : Ch()\}, y : \{b : Ch()\}, \Gamma$, where abstracting on $x@y$ is not possible since y would still be free in the conclusion. This term is typable in the system of [11], with the type $Ch(loc\{b : Ch()\}[Ch()])$ for a (this example is due to Matthew Hennessy). We could have used more general patterns of compound names, like $\{c_1, \ldots, c_n\}@\ell$, but the previous counter-example would still hold, because a process receiving such a compound name can only use $c_1, \ldots, c_n$ at ℓ, and no other name.

Our rules for systems of located processes, given in the Figure 6, are essentially the same as the ones of [11]. Let us see how to type the client of the RPC example, assuming that the data and results are of value types, and assuming that the continuation P is typable in the context $\overrightarrow{v : val}, \Gamma$ at the location ℓ of the client. Let $\psi = \{a : Ch(\overrightarrow{val}, Ch(\overrightarrow{val})^{@})\}$ and $\phi = \{r : Ch(\overrightarrow{val})\}$, and $\Delta = \ell' : \psi\,,\ \ell : \phi\,,\ \overrightarrow{d : val}\,,\ \Gamma$ – assuming that this is a legal context. Then we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ell : \phi\,,\ \overrightarrow{d : val} \vdash_{\ell'} \overrightarrow{d : val}\,,\ r@\ell : Ch(\overrightarrow{val})^{@}}}{\ell' : \psi\,,\ \ell : \phi\,,\ \overrightarrow{d : val} \vdash_{\ell'} \overline{a}(\vec{d}, r@\ell)}}{\ell' : \psi\,,\ \ell : \phi\,,\ \overrightarrow{d : val} \vdash_{\ell} \mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell)}}{\Delta \vdash_{\ell} \mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell)} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\overrightarrow{v : val}, \Gamma \vdash_{\ell} P} \qquad \dfrac{}{\overrightarrow{v : val} \vdash_{\ell} \overrightarrow{v : val}}}{\ell : \phi, \Gamma \vdash_{\ell} r(\vec{v}).P}}{\Delta \vdash_{\ell} r(\vec{v}).P}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vdash_{\ell} (\mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell) \mid r(\vec{v}).P)}}{\ell' : \psi\,,\ \overrightarrow{d : val}\,,\ \Gamma \vdash_{\ell} (\nu r)(\mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell) \mid r(\vec{v}).P)}}{\ell' : \psi\,,\ \overrightarrow{d : val}\,,\ \Gamma \vdash [\ell :: (\nu r)(\mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell) \mid r(\vec{v}).P)]}$$

The RPC example illustrates a situation where a request is moving towards a service. One may wonder what happens for a process migrating out of its current location, where some ressource is available. For instance, to type $(\nu a)(a(\vec{u}).R \mid \mathsf{go}\,\ell.P \mid Q)$ at the current locality ℓ' requires, by virtue of the rules for scoping, input, parallel composition and migration, to type P at ℓ with a typing context containing $\ell' : \{a : Ch(\vec{\tau})\}$ as the only assumption about a. This means that in P the name a can only be used at ℓ', that is, if $\ell' \neq \ell$, either in a compound name $a@\ell'$ or inside a guard $\mathsf{go}\,\ell'$. Similarly, in the case of a moving "server", like in $(\nu a)(\mathsf{go}\,\ell.a(\vec{u}).R \mid P)$, one can see that the process P must use the name a at ℓ, again in a compound name (unless ℓ is the current locality), or inside a migration construct $\mathsf{go}\,\ell$. Now let us see an example where access to a public or private "resource" is exported from a locality to another. More specifically, we consider a distributed system which at locality ℓ exports (privately) its location name to a distinct locality ℓ', while at ℓ' it learns the name ℓ and uses it to move and ask some service. Such a system may be for instance

$$S = (\nu \ell : \{b : Ch()\})[\ell :: b^*().R \mid \overline{a}@\ell'(\ell)] \ \mid\ [\ell' :: a(\ell).\overline{b}@\ell()]$$

Let us see how to type such a term. If we let $\gamma = Ch()$, $\psi = \{b : \gamma\}$ and $\delta = Ch(\psi)$, then the following inference is valid

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ell : \psi \vdash_{\ell} \overline{b}()}}{\ell : \psi \vdash_{\ell'} \mathsf{go}\,\ell.\overline{b}()} \qquad \dfrac{}{\ell : \psi \vdash_{\ell'} \ell : \psi}}{\ell' : \{a : \delta\} \vdash_{\ell'} a(\ell).\mathsf{go}\,\ell.\overline{b}()}}{\ell' : \{a : \delta\} \vdash [\ell' :: a(\ell).\overline{b}@\ell()]}$$

We also have, assuming some typing for $b^*().R$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\ell : \psi\,,\, \Gamma \vdash_\ell b^*().R}}{\Delta \vdash_\ell b^*().R} \qquad \dfrac{\dfrac{\dfrac{}{\ell : \psi \vdash_{\ell'} \ell : \psi}}{\ell' : \{a : \delta\}\,,\, \ell : \psi \vdash_{\ell'} \overline{a}(\ell)}}{\ell' : \{a : \delta\}\,,\, \ell : \psi \vdash_\ell \mathsf{go}\,\ell'.\overline{a}(\ell)}}{\Delta \vdash_\ell \mathsf{go}\,\ell'.\overline{a}(\ell)}}{\Delta \vdash_\ell b^*().R \mid \mathsf{go}\,\ell'.\overline{a}(\ell)}}{\Delta \vdash [\ell :: b^*().R \mid \mathsf{go}\,\ell'.\overline{a}(\ell)]}}{\ell' : \{a : \delta\}\,,\, \Gamma \vdash (\nu \ell : \psi)[\ell :: b^*().R \mid \overline{a}@\ell'(\ell)]}$$

where $\Delta = \ell' : \{a : \delta\}\,,\, \ell : \psi\,,\, \Gamma$. We conclude that the system S agrees with the typing context $\ell' : \{a : \delta\}\,,\, \Gamma$. In this example, the "resource" $b^*().R$ is publicly known, outside its own location. One can imagine a variant of this situation, where the object is only known within its locality. Then one should export the name of this resource together with its location, and this may be written:

$$S' = [\ell :: (\nu b)(b^*().R \mid \overline{a}@\ell'(b@\ell))\,] \;\mid\; [\ell' :: a(b@\ell).\overline{b}@\ell()]$$

We let the reader check that this term is typable with $\ell' : \{a : Ch(\gamma^{@})\}$.

Our main result about typing is the standard "subject reduction property". This result is needed to establish the message deliverability property. Although a similar result is established in [11], we give a proof here, since our calculus is slightly different, especially regarding substitution. To prove that typing is preserved by reduction, we need some auxiliary results, mainly relating substitution and typing, which are stated and proved in the Appendix. Let us denote by $P \sqsubseteq_{\mathcal{T}} Q$ the fact that Q has more typings than P. Formally, the relation $\sqsubseteq_{\mathcal{T}}$ is defined by

$$\begin{aligned}
P \sqsubseteq_{\mathcal{T}} Q &\;\Leftrightarrow\; \forall \Gamma\, \forall \ell\, (\Gamma \vdash_\ell P \;\Rightarrow\; \Gamma \vdash_\ell Q) \\
T \sqsubseteq_{\mathcal{T}} T' &\;\Leftrightarrow\; \forall \Gamma\, \forall \ell\, \forall \tau (\Gamma \vdash_\ell T : \tau \;\Rightarrow\; \Gamma \vdash_\ell T' : \tau) \\
S \sqsubseteq_{\mathcal{T}} S' &\;\Leftrightarrow\; \forall \Gamma\, (\Gamma \vdash S \;\Rightarrow\; \Gamma \vdash S')
\end{aligned}$$

LEMMA 3.2. *The relation $\sqsubseteq_{\mathcal{T}}$ is a precongruence, that is a preorder compatible with the syntactic constructs.*

PROOF: this is obvious, since the typing of a term only depends on the typing of its components. ❑

We denote by $=_{\mathcal{T}}$ the associated equivalence relation, that is $U =_{\mathcal{T}} V$ if and only if $U \sqsubseteq_{\mathcal{T}} V$ and $V \sqsubseteq_{\mathcal{T}} U$. This is obviously a congruence. The first step towards "subject reduction" is to show that typing is preserved by structural equivalence.

PROPOSITION 3.3. *The relation $\sqsubseteq_{\mathcal{T}}$ contains the structural equivalence relation $\equiv$.*

PROOF: by the lemmas 3.2 and A.5, the relation $=_{\mathcal{T}}$ is a congruence containing α-conversion, and therefore it is enough to show that for each axiom $U \equiv V$ we have $U \sqsubseteq_{\mathcal{T}} V$ and $V \sqsubseteq_{\mathcal{T}} U$. This is trivial for the neutrality of $\mathbf{0}$, the associativity and commutativity of parallel composition, and for the "routing" axiom $[\ell :: P \mid Q] \equiv [\ell :: P] \mid [\ell :: Q]$.

Let $\Gamma \vdash_\ell ((\nu w)P \mid Q)$ with $\mathsf{subj}(w) \notin \mathsf{fn}(Q)$. By the Lemma A.5 we may assume that $\mathsf{subj}(w)$ does not occur in Γ. The proof of this sequent has the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x:\tau\,,\,\Delta' \vdash_\ell P}}{\Delta' \vdash_\ell (\nu w)P}}{\dfrac{\vdots}{\Delta \vdash_\ell (\nu w)P}}\ \Delta' \subseteq \Delta \qquad \dfrac{\vdots}{\Delta \vdash_\ell Q}}{\dfrac{\Delta \vdash_\ell ((\nu w)P \mid Q)}{\dfrac{\vdots}{\Gamma \vdash_\ell ((\nu w)P \mid Q)}}}$$

where $\Delta \subseteq \Gamma$ and $x:\tau$ is respectively $\ell:\{a:\gamma\}$ if $w = a$, or $\ell':\{a:\gamma\}$ if $w = a@\ell'$, or $a:val$ if $w = a:val$. Since $\mathsf{subj}(w)$ does not occur in Δ, the context $x:\tau, \Delta$ is well-defined, and by weakening we also have $x:\tau\,,\,\Delta \vdash_\ell Q$, therefore $\Gamma \vdash_\ell (\nu w)(P \mid Q)$. Conversely, a proof of $\Gamma \vdash_\ell (\nu w)(P \mid Q)$ has the following form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Delta \vdash_\ell P} \qquad \dfrac{\vdots}{\Delta \vdash_\ell Q}}{\Delta \vdash_\ell P \mid Q}}{\dfrac{\vdots}{x:\tau\,,\,\Gamma' \vdash_\ell P \mid Q}}}{\Gamma' \vdash_\ell (\nu w)(P \mid Q)}}{\dfrac{\vdots}{\Gamma \vdash_\ell (\nu w)(P \mid Q)}}$$

with $\Delta \subseteq x:\tau, \Gamma'$ and $\Gamma' \subseteq \Gamma$. If $\Delta = x:\tau, \Delta'$ then by the Lemma A.2 we have $\Delta' \vdash_\ell Q$, and $\Delta' \vdash_\ell (\nu w)P$, therefore $\Gamma \vdash_\ell ((\nu w)P \mid Q)$. Otherwise, $\Delta \subseteq \Gamma$, and by weakening $x:\tau, \Delta \vdash_\ell P$. From this we conclude $\Delta \vdash_\ell (\nu w)P$, hence $\Gamma \vdash_\ell ((\nu w)P \mid Q)$. The proof is similar for the case of $((\nu w)S \mid S') \equiv (\nu w)(S \mid S')$.

For $[\ell :: (\nu w)P] \equiv (\nu w@\ell)[\ell :: P]$, let us assume that $\Gamma \vdash [\ell :: (\nu w)P]$. The proof of this sequent has the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x:\tau\,,\,\Delta \vdash_\ell P}}{\Delta \vdash_\ell (\nu w)P}}{\dfrac{\vdots}{\Gamma' \vdash_\ell (\nu w)P}}}{\Gamma' \vdash [\ell :: (\nu w)P]}}{\dfrac{\vdots}{\Gamma \vdash [\ell :: (\nu w)P]}}$$

with $\Delta \subseteq \Gamma' \subseteq \Gamma$, and where $x : \tau$ is respectively $\ell : \{a : \gamma\}$ if $w = a$, or $\ell' : \{a : \gamma\}$ if $w = a@\ell'$, or $a : val$ if $w = a : val$. Then it is easy to see that the following is a valid inference:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : \tau\,,\ \Delta \vdash_\ell P}}{x : \tau\,,\ \Delta \vdash [\ell :: P]}}{\Delta \vdash (\nu w@\ell)[\ell :: P]}}{\Gamma \vdash (\nu w@\ell)[\ell :: P]}$$

For the symmetric case where $\Gamma \vdash (\nu w@\ell)[\ell :: P]$ the proof is similar. ⬓

Now we can prove the main result of this section.

THEOREM (PRESERVATION of TYPING) 3.4. $U \to V \ \Rightarrow\ U \sqsubseteq_{\mathcal{T}} V$

PROOF: by induction on the definition of $U \to V$. Since $\sqsubseteq_{\mathcal{T}}$ is a precongruence containing the structural equivalence relation, it is enough to consider the axioms of reduction. The cases of $[\ell' :: \mathsf{go}\,\ell.P] \to [\ell :: P]$, $[a = a]P, Q \to P$ and $[a = b]P, Q \to Q$ are trivial.

For the case of the communication law $(\overline{a}(\vec{v}) \mid a(\vec{u}).P) \to [\vec{v}/\vec{u}]P$, let Γ be a context such that $\Gamma \vdash_\ell (\overline{a}(\vec{v}) \mid a(\vec{u}).P)$. Then the proof of this sequent has the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Theta \vdash_\ell \overrightarrow{v : \theta}}}{\ell : \{a : Ch(\vec{\theta})\}\,,\ \Theta \vdash_\ell \overline{a}(\vec{v})}}{\dfrac{\vdots}{\Delta \vdash_\ell \overline{a}(\vec{v})}} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Sigma\,,\ \Delta' \vdash_\ell P} \qquad \dfrac{\vdots}{\Sigma \vdash_\ell \overrightarrow{u : \tau}}}{\ell : \{a : Ch(\vec{\tau})\}\,,\ \Delta' \vdash_\ell a(\vec{u}).P}}{\dfrac{\vdots}{\Delta \vdash_\ell a(\vec{u}).P}}}{\Delta \vdash_\ell \overline{a}(\vec{v}) \mid a(\vec{u}).P}}{\dfrac{\vdots}{\Gamma \vdash_\ell \overline{a}(\vec{v}) \mid a(\vec{u}).P}}$$

with $\ell : \{a : Ch(\vec{\theta})\}, \Theta \subseteq \Delta \subseteq \Gamma$ and $\ell : \{a : Ch(\vec{\tau})\}\,,\ \Delta' \subseteq \Delta$, and no name of $\mathsf{nm}(\vec{u})$ occurs in $\ell : \{a : Ch(\vec{\tau})\}\,,\ \Delta'$, (and these names are distinct from ℓ). The names of $\mathsf{nm}(\vec{u})$ do not occur either in $\vec{\tau}$, and therefore $(\vec{v}/\vec{u})\Sigma$ is a typing context. Then by the Substitution Lemma A.3(ii), the sequent

$$(\vec{v}/\vec{u})\Sigma\,,\ \Delta' \vdash_\ell [\vec{v}/\vec{u}]P$$

is provable. Since Δ is a typing context, we must have $\vec{\theta} = \vec{\tau}$, and then by the Lemma A.3(i) and the Remark 3.1 we have $\Theta = (\vec{v}/\vec{u})\Sigma$, therefore the sequent $\Gamma \vdash_\ell [\vec{v}/\vec{u}]P$ is provable.

$$\frac{}{\Vdash \overline{a}\vec{u}} \qquad \frac{a \Vdash P}{a \Vdash a(\vec{u}).P} \qquad \frac{I \Vdash U \ , \ I' \Vdash V}{I \, , \, I' \Vdash (U \mid V)} \ I \cap I' = \emptyset$$

$$\frac{u, I \Vdash U}{I \Vdash (\nu u)U} \qquad \frac{I \Vdash U}{I \Vdash (\nu a : val)U} \qquad \frac{\mathsf{dom}(\psi)@\ell \, , \, I \Vdash S}{I \Vdash (\nu \ell : \psi)S}$$

$$\frac{I \Vdash P \ , \ I \Vdash Q}{I \Vdash [a = b]P, Q} \qquad \frac{}{\Vdash A} \qquad \frac{a \Vdash P}{\Vdash (\mathsf{rec}\, A(a, \vec{u}).P)} \qquad \frac{}{a \Vdash T(a, \vec{u})}$$

$$\frac{I \Vdash P}{I@\ell \Vdash \mathsf{go}\, \ell.P} \ \{ a \mid a, a@\ell \in I \} = \emptyset \qquad \frac{I \Vdash P}{I@\ell \Vdash [\ell :: P]} \ \{ a \mid a, a@\ell \in I \} = \emptyset$$

Figure 7: Well-formed terms

Now let us consider the unfolding, that is the case of $(\mathsf{rec}\, A(\vec{u}).P)(\vec{v}) \rightarrow [\mathsf{rec}\, A(\vec{u}).P/A][\vec{v}/\vec{u}]P$, and assume that the sequent $\Gamma \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P)(\vec{v})$ is provable. The proof of this sequent has the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A : Ch(\vec{\tau}) \, , \, \Sigma \, , \, \Delta' \vdash_\ell P} \qquad \dfrac{\vdots}{\Sigma \vdash_\ell \overrightarrow{u : \tau}}}{\Delta' \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P) : Ch(\vec{\tau})}}{\dfrac{\dfrac{\vdots}{\Delta \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P) : Ch(\vec{\tau})} \qquad \dfrac{\vdots}{\Theta \vdash_\ell \overrightarrow{v : \tau}}}{\Delta \, , \, \Theta \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P)(\vec{v})}}}{\dfrac{\vdots}{\Gamma \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P)(\vec{v})}}$$

with $\Delta, \Theta \subseteq \Gamma$ and $\Delta' \subseteq \Delta$. As in the previous case, the sequent $A : Ch(\vec{\tau}) \, , \, \Theta \, , \, \Delta' \vdash_\ell [\vec{v}/\vec{u}]P$ is provable. By the Lemma A.5 we may assume that there is no name which is both free and bound in P, so that the substitution $[(\mathsf{rec}\, A(\vec{u}).P)/A][\vec{v}/\vec{u}]P$ is defined. Since $\Delta' \vdash_\ell (\mathsf{rec}\, A(\vec{u}).P) : Ch(\vec{\tau})$, the sequent

$$\Theta \, , \, \Delta' \vdash_\ell [(\mathsf{rec}\, A(\vec{u}).P)/A][\vec{v}/\vec{u}]P$$

is provable, by the Substitution Lemma A.3(iv), hence also $\Gamma' \vdash_\ell [(\mathsf{rec}\, A(\vec{u}).P)/A][\vec{v}/\vec{u}]P$. ❑

4. Interfaces and Receptive Processes.

It is easily seen that the type system does not guarantee the kind of safety property we are looking for: we can obviously type non-receptive processes, and we can also type processes sending a message that will never find a corresponding receiver, like for instance $(\nu a)\overline{a}()$. Then we introduce an inference system for checking "well-formedness" of processes. Basically, to be well-formed a process must not contain nested inputs on different names. Moreover, any input must be involved in a recursive process that makes the receiver reincarnate itself, possibly in a different state, after being consumed. In addition, we shall impose, as in π_1 [2], that there is a unique receiver for each name, that is, two parallel components of a process are not allowed to receive on a same name. Last but not least, we demand that to restrict the scope of a name of channel type, we know for a fact that a resource is provided – that is, a receiver exists – for that name.

The well-formed processes, and more generally the well-formed terms, are defined as the ones for which a statement $I \Vdash U$, that is "*U is well-formed with interface I*", can be proved. In this statement the interface I is a finite set of names on which a process may perform an input. We present the rules for well-formedness

in Figure 7, where we use the same convention as for the typing regarding the binding constructs, and where, as usual, a set $I = \{u_1, \ldots, u_n\}$ is represented as the list $u_1, \ldots, u_n$ of its elements, and union is written I, I'.

Let us comment on these rules. In the rule for input $a(\vec{u}).P$, the interface of the continuation P is assumed to be $\{a\}$, that is, P appears to offer inputs on a, and only on that name. Notice that received names are not in the interface of P, since $a \notin \mathsf{nm}(\vec{u})$. Therefore a well-formed process satisfies the constraint that a received name must not be used as an input channel, a constraint that characterizes the "local" (asynchronous, polyadic) π-calculus of [14]. Typically, $a(c).c(\vec{d}).P$ is not well-formed. Assuming "locality" is important for our purpose, since otherwise the environment of a process like $(\nu a)(\overline{a}(\vec{u}) \mid \overline{b}(a))$ could create a receiver for a, and message deliverability would become more difficult to check. Note also that we only accept $(\nu w)P$ either if there is a receiver on $\mathsf{subj}(w)$ in P – in which case there is again an implicit side condition $\mathsf{subj}(w) \notin \mathsf{nm}(I)$ – or if w is a typed value or location name. In the latter case, where $w = \ell : \psi$ (and $\ell \notin \mathsf{nm}(I)$), if $\psi = \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$, then we require a receiver to be provided at ℓ for each name $a_1, \ldots, a_n$ (recall that $\mathsf{dom}(\psi)@\ell = \{a_1@\ell, \ldots, a_n@\ell\}$). Then the location type ψ describes the set of resources available to an agent at the location ℓ. Observe also that the only way to introduce a name a in the interface is to introduce a process identifier applied to a, that is $A(a, \vec{u})$, which will be involved in a guarded recursion, where the input guarding the recursive call must be on the same name a – thanks to the rule for input.

The rule for parallel composition in Figure 7 could be relaxed to allow the intersection of I and I' to be non-empty – and simultaneously we would not impose any condition for $I@\ell \Vdash \mathsf{go}\,\ell.P$ and $I@\ell \Vdash [\ell :: P]$. In this case, we can still guarantee that messages are deliverable although not necessarily to a unique receiving process. We require the uniqueness of the receivers because it seems a sensible assumption both from a specification and an implementation point of view (*cf.* object-oriented models, and [9]). Moreover, this property is useful to establish that we do not lose expressive power, and more specifically to establish that we can encode the π_1-calculus, see the last section of the paper.

Let us see some examples, starting with our running RPC example. The client process, that is $Q = (\nu r)(\,\mathsf{go}\,\ell'.\overline{a}(\vec{d}, r@\ell) \mid r(\vec{v}).P\,)$ is only well-formed if r is in the interface of P. This may seem somewhat unnatural, since r is a return channel which, intuitively, is supposed to receive just one answer from the called procedure. Indeed, it would be better to incorporate the treatment of *linear channels* as in [13], but we leave this for further work. There is actually a standard way to turn a process like the continuation P which is not supposed to be receptive on r into a receptive process. To see this, let us first introduce some notations. The *identity* agent is the following:

$$Id_a \;=_{\mathrm{def}}\; \mathsf{rec}\,A(a).a(\vec{u}).(\overline{a}(\vec{u}) \mid A(a)) = a^*(\vec{u}).\overline{a}(\vec{u})$$

(recall our convention that $\mathsf{rec}\,A(\vec{u}).P$ stands for $(\mathsf{rec}\,A(\vec{u}).P)(\vec{u})$). This process, which repeatedly waits for inputs on a and sends them back, is well-formed, with interface $\{a\}$, and has the typing $\ell : \{a : Ch(\vec{\tau})\} \vdash_\ell Id_a$. As we shall see, this term is just as good as the process that repeatedly receives messages on a and ignores them:

$$T_a \;=_{\mathrm{def}}\; \mathsf{rec}\,A(a).a(\vec{u}).A(a)$$

This agent, which is more or less $a^*(\vec{u}).\mathbf{0}$, has the same typing and interface as Id_a. We let the reader see how to type this one, which has interface $\{a\}$. Now we recall the *input once* construct of [2], given by

$$a(\vec{u}){:}P \;=_{\mathrm{def}}\; a(\vec{u}).(P \mid T_a)$$

It is easy to see that this term has the same typing as $a(\vec{u}).P$, and that $a \Vdash a(\vec{u}){:}P$ if $\Vdash P$. This could actually be defined using Id_a instead of T_a. Then we can turn the RPC client into a well-formed process – provided that the continuation P has an empty interface – by replacing the ordinary input with an input once:

$$Q = (\nu r)(\,\overline{a}@\ell'(\vec{d}, r@\ell) \mid r(\vec{v}){:}P\,)$$

which has the same typing as before. Regarding the remote procedure, one can see that the replicated input $a^*(\vec{u}).P$ has the same typing and well-formedness as the input once construct – that is the same typing as input, and $a \Vdash a^*(\vec{u}).P$ if $\Vdash P$.

We have seen that the typing discipline imposes some constraints on the usage of names, as regards the locality where they can be used. One can see that well-formedness imposes some constraints too, enforcing the fact that the interface must be persistent in a sense. For instance, a term like $a(\vec{u}).(P \mid \mathsf{go}\,\ell.a(\vec{v}).Q)$ is

not acceptable, because the subterm $(P \mid \mathsf{go}\,\ell.a(\vec{v}).Q)$ should have interface $\{a\}$, while it is only acceptable with an interface containing $a@\ell$. This means that servers are not so easily moved, which is quite natural, given the properties we are looking for. Indeed, we do not want the receiver to disappear going elsewhere after having received a message. We shall see later how to manage migrating resources.

We may observe that for the terms of the ordinary π-calculus, we also get a notion of well-formedness $I \Vdash P$ where I is a set of simple names, simply by forbidding the rules for $\mathsf{go}.\ell.P$. The system we get is a refinement of the one introduced for the π_1-calculus in [2]. Now we prove our main result, that is the message deliverability property. We will need first to prove a subject reduction property for well-formedness, and a receptiveness property. The following should be clear:

REMARK 4.1. *If $I \Vdash U$ is provable then $\mathsf{nm}(I) \subseteq \mathsf{fn}(U)$.*

Again we need to establish a property relating well-formedness and substitution. In the following lemma we use a notation $\mathsf{S}(I \Vdash U)$ similar to the one we used regarding typing.

LEMMA 4.2. *Let $I \Vdash U$ and S be a substitution such that $[\mathsf{S}]U$ is defined, and S is injective on I. Then $\mathsf{S}(I \Vdash U)$, and if $I \Vdash [\mathsf{S}]U$ then $\mathsf{S}^{-1}(I) \Vdash U$. If $I \Vdash P$ and $\Vdash T$, and $\mathsf{fn}(T) \cap \mathsf{bn}(P) = \emptyset$ then $I \Vdash [T/A]P$.*

PROOF: by induction on the inference of $I \Vdash U$, trivial. ▭

The relation $\sqsubseteq_{\mathcal{R}}$ given by

$$U \sqsubseteq_{\mathcal{R}} V \;\Leftrightarrow\; \forall I\,(I \Vdash U \;\Rightarrow\; I \Vdash V)$$

is clearly a precongruence. Let $=_{\mathcal{R}}$ be the associated equivalence. We have:

COROLLARY 4.3. *The relation $=_{\mathcal{R}}$ contains the relation $=_\alpha$ of α-conversion.*

PROOF: by induction on the definition of $=_\alpha$, easy. ▭

LEMME 4.4. *The relation $\sqsubseteq_{\mathcal{R}}$ contains the structural equivalence relation $\equiv$.*

PROOF: since $\sqsubseteq_{\mathcal{R}}$ is a precongruence containing the relation of α-conversion, it is enough to show that for each axiom $U \equiv V$ we have $U \sqsubseteq_{\mathcal{R}} V$ and $V \sqsubseteq_{\mathcal{R}} U$. This is trivial for the neutrality of $\mathbf{0}$, the associativity and commutativity of parallel composition, and for the "routing" axiom $[\ell :: P \mid Q] \equiv [\ell :: P] \mid [\ell :: Q]$.

Let $I \Vdash ((\nu w)U \mid V)$ with $\mathsf{subj}(w) \notin \mathsf{fn}(V)$. By the Corollary 4.3 we may assume that $\mathsf{subj}(w)$ does not occur in I. Let us examine the case where $w = u$. We have

$$\dfrac{\dfrac{\dfrac{\vdots}{u\,,\, I_0 \Vdash U}}{I_0 \Vdash (\nu u)U} \qquad \dfrac{\vdots}{I_1 \Vdash V}}{I_0\,,\, I_1 \Vdash ((\nu u)U \mid V)}\; I_0 \cap I_1 = \emptyset$$

where $I_0, I_1 = I$. By the Remark 4.1 we have $u \notin I_1$, therefore $I \Vdash (\nu u)(U \mid V)$. The other cases are similar, as well as the symmetric case where the hypothesis is $I \Vdash (\nu w)(U \mid V)$. Let us now examine the case where $I \Vdash [\ell :: (\nu u)P]$. We have

$$\dfrac{\dfrac{\dfrac{\vdots}{u\,,\, I' \Vdash P}}{I' \Vdash (\nu u)P}}{I \Vdash [\ell :: (\nu u)P]}\; \{\, a \mid a, a@\ell \in I' \,\} = \emptyset$$

where $\mathsf{subj}(u) \notin \mathsf{nm}(I')$ and $I = I'@\ell$. Then we also have $\{\, a \mid a, a@\ell \in u, I' \,\} = \emptyset$, and therefore the following is a valid inference:

$$\dfrac{\dfrac{\dfrac{\vdots}{u\,,\, I' \Vdash P}}{u@\ell\,,\, I'@\ell \Vdash [\ell :: P]}}{I \Vdash (\nu u@\ell)[\ell :: P]}$$

The symmetric case, where the hypothesis is $I \Vdash (\nu u@\ell)[\ell :: P]$, is easy. ▭

PROPOSITION (PRESERVATION of WELL-FORMEDNESS) 4.5. $U \to V \;\Rightarrow\; U \sqsubseteq_{\mathcal{R}} V$

PROOF: by induction on the definition of $U \to V$. It is enough to consider the axioms of reduction. The cases of $[\ell' :: \mathsf{go}\,\ell.P] \to [\ell :: P]$, $[a = a]P, Q \to P$ and $[a = b]P, Q \to Q$ are trivial. For the case of $(\overline{a}(\vec{v}) \mid a(\vec{u}).P) \to [\vec{v}/\vec{u}]P$, we have

$$\dfrac{\dfrac{}{\Vdash \overline{a}(\vec{v})} \qquad \dfrac{\dfrac{\vdots}{a \Vdash P}}{a \Vdash a(\vec{u}).P}}{a \Vdash \overline{a}(\vec{v}) \mid a(\vec{u}).P}$$

with $a \notin \mathsf{nm}(\vec{u})$. Then by the Lemma 4.2 we also have $a \Vdash [\vec{v}/\vec{u}]P$. For the unfolding case, that is $T(\vec{v}) \to [T/A][\vec{v}/\vec{u}]P$ where $T = (\mathsf{rec}\, A(\vec{u}).P)$, we have

$$\dfrac{\dfrac{\dfrac{\vdots}{a' \Vdash P}}{\Vdash (\mathsf{rec}\, A(a', \vec{u'}).P)}}{a \Vdash (\mathsf{rec}\, A(a', \vec{u'}).P)(a, \vec{v'})}$$

where $\vec{u} = a', \vec{u'}$ and $\vec{v} = a, \vec{v'}$, and we conclude again by the Lemma 4.2. ▭

The receptiveness property states that if a name is in the interface of a process, then this process offers a resource (a receiver) for that name. This resource may not be immediately available, but one shows that some normalizing computation will reveal it. To express this formally, let us define the predicates $U \downarrow u$ and $U \not\downarrow u$ inductively as follows – recall that $U \twoheadrightarrow V$ is reduction without communication:

$$\frac{}{a(\vec{u}).P \downarrow a} \qquad \frac{U \downarrow u}{(U \mid V) \downarrow u} \qquad \frac{V \downarrow u}{(U \mid V) \downarrow u} \qquad \frac{U \downarrow u}{(\nu w)U \downarrow u}\ \mathsf{nm}(u) \cap \mathsf{subj}(w) = \emptyset$$

$$\frac{S \downarrow a@\ell}{(\nu \ell' : \psi)S \downarrow a@\ell}\ a \neq \ell' \neq \ell \qquad \frac{P \not\downarrow u}{\mathsf{go}\,\ell.P \downarrow u@\ell} \qquad \frac{P \downarrow u}{[\ell :: P] \downarrow u@\ell} \qquad \frac{U \overset{*}{\twoheadrightarrow} V \ ,\ V \downarrow u}{U \not\downarrow u}$$

The only non-standard clause in this definition is the one for $\mathsf{go}\,\ell.P$: although this is a process that does not reduce, we must anticipate the fact that after having migrated, P will offer some input. It is easy to see that if $U \downarrow u$ and $V \equiv U$ then $V \downarrow u$, and that

$$\begin{array}{rcll} P \downarrow a & \Leftrightarrow & P \equiv (\nu \vec{w})(a(\vec{u}).R \mid Q) & a \notin \mathsf{subj}(\vec{w}) \\ S \downarrow a@\ell & \Leftrightarrow & S \equiv (\nu \vec{w})([\ell' :: P] \mid S') & P \downarrow v,\ v@\ell' = a@\ell \text{ and } a, \ell \notin \mathsf{subj}(\vec{w}) \end{array}$$

The following lemma states that, after a preliminary phase where some migration can occur, all the receivers that are exhibited in a network are immediately available.

LEMMA 4.6. *If $P \downarrow a@\ell$ then there exists P' such that $P' \downarrow a$ and $[\ell' :: P] \overset{*}{\twoheadrightarrow} (\nu \vec{w})([\ell :: P'] \mid S)$ for some S, with $a \notin \mathsf{subj}(\vec{w})$.*

PROOF: by induction on the definition of $P \downarrow a@\ell$. If $P = \mathsf{go}\,\ell''.Q$ with $Q \not\downarrow u$ and $u@\ell'' = a@\ell$, we have $[\ell' :: P] \twoheadrightarrow [\ell'' :: Q]$, and by definition there exists P' such that $Q \overset{*}{\twoheadrightarrow} P'$ and $P' \downarrow u$. Then we have $[\ell' :: P] \overset{*}{\twoheadrightarrow} [\ell'' :: P']$, and there are two cases: if $u = a$ and $\ell'' = \ell$, then we are done; if $u = a@\ell$ then we use the induction hypothesis for $[\ell'' :: P']$. If $P = (Q \mid R)$ with, for instance, $Q \downarrow a@\ell$ then $[\ell' :: P] \equiv [\ell' :: Q] \mid [\ell' :: R]$, and we use the induction hypothesis. If $P = (\nu w)Q$ with $Q \downarrow a@\ell$ and $a \notin \mathsf{subj}(w)$ we have $[\ell' :: P] \equiv (\nu w@\ell')[\ell' :: Q]$, and we use the induction hypothesis. ▭

PROPOSITION (RECEPTIVENESS) 4.7. *Let U be a well-formed, typable term. Then:*
(i) *if $I \Vdash U$ then $u \in I \Leftrightarrow U \not\downarrow u$,*
(ii) *if $U \not\downarrow u$ and $U \to V$ then $V \not\downarrow u$.*

PROOF:

(i)($\Rightarrow$) we prove a more general statement, namely that $u, I \Vdash U \Rightarrow [\mathsf{S}]U \not\downarrow \mathsf{S}(u)$ for any substitution S such that $[\mathsf{S}]U$ is defined. We proceed by induction on U. Most of the cases are trivial (using the laws of structural equivalence); let us only examine the ones where some reduction is needed (notice that in the case of $\mathsf{go}\,\ell.P$ we see why a non standard rule for $\downarrow u$ is needed). If U is $[b = c]Q, R$, then the inference of $u, I \Vdash U$ has the form

$$\frac{\dfrac{\vdots}{u\,,\, I \Vdash Q} \qquad \dfrac{\vdots}{u\,,\, I \Vdash R}}{u\,,\, I \Vdash [b = c]Q, R}$$

then one has $[\mathsf{S}]([b = c]Q, R) = [\mathsf{S}(b) = \mathsf{S}(c)][\mathsf{S}]Q, [\mathsf{S}]R$, and therefore $[\mathsf{S}]([b = c]Q, R) \rightarrow [\mathsf{S}]Q$ or $[\mathsf{S}]([b = c]Q, R) \rightarrow [\mathsf{S}]R$, and one uses the induction hypothesis. If $U = T(a, \vec{u})$ and $I = \emptyset$, we have $T = (\mathsf{rec}\, A(b, \vec{v}).Q)$ since U is guarded, and $\vec{v}$ and $\vec{u}$ have the same length since U is typable. Then the inference $a \Vdash U$ has the form

$$\frac{\dfrac{\dfrac{\vdots}{b \Vdash Q}}{\Vdash (\mathsf{rec}\, A(b, \vec{v}).Q)}}{a \Vdash (\mathsf{rec}\, A(b, \vec{v}).Q)(a, \vec{u})}$$

We have $[\mathsf{S}]((\mathsf{rec}\, A(b, \vec{v}).Q)(a, \vec{u})) = T'(c, \vec{w})$ with $T' = (\mathsf{rec}\, A(b, \vec{v}).[\mathsf{S}']Q)$ where $\mathsf{S}' = \mathsf{S}\restriction \mathsf{dom}(\mathsf{S}) - \{b\} \cup \mathsf{nm}(\vec{v})$, and $(c, \vec{w}) = [\mathsf{S}](a, \vec{u})$, and therefore

$$[\mathsf{S}]((\mathsf{rec}\, A(b, \vec{v}).Q)(a, \vec{u})) \rightarrow [T'/A][c, \vec{w}/b, \vec{v}][\mathsf{S}']Q$$

Clearly Q is guarded and typable, and therefore by induction hypothesis $[c, \vec{w}/b, \vec{v}][\mathsf{S}']Q \not\downarrow c$, hence also $[\mathsf{S}]((\mathsf{rec}\, A(b, \vec{v}).Q)(a, \vec{u})) \not\downarrow \mathsf{S}(a)$ by the Lemma A.1.

(i)($\Leftarrow$) we prove that $I \Vdash U$ and $U \downarrow u$ or $U \not\downarrow u$ implies $u \in I$, by induction on the definition of the predicates $U \downarrow u$ and $U \not\downarrow u$. The only case that deserves some consideration is $U \not\downarrow u$ with $U \stackrel{*}{\rightarrow\!\!\!\!\!\rightarrow} V$ and $V \downarrow u$. By the Proposition 4.5 we have $I \Vdash V$ (and V is typable by the Proposition 3.4), and by induction hypothesis this implies $u \in I$.

(ii) assume that $I \Vdash U$. Then by the previous point $u \in I$. By the Propositions 3.4 and 4.5, V is typable and $I \Vdash V$, which, by (i)($\Rightarrow$), implies $V \not\downarrow u$. ⬙

An immediate consequence of this proposition is that we statically know the locality of each receiver in a term: for a public name, it is indicated in the interface – the locality of a receiver on a is the current one if $a \in I$, and is ℓ if $a@\ell \in I$ (as we shall see with the "distributed object" example below, the same channel may have several receivers, though in different locations). For a subterm $(\nu u)P$ of a well-formed process, we know that u is in the interface of P. Note however that in $(\nu a@\ell)P$, the location name ℓ is free, and may be received as a parameter from a message, see our last example of a "migrating cell" in the next section.

This result suggests the denomination "*distributed (asynchronous) receptive π-calculus, with unique receivers*", in short the $\mathrm{D}\pi_1^r$-calculus, for the set of well-formed closed processes, which is closed by reduction. Similarly we call π_1^r, that is "*the receptive π_1-calculus*", the sub-calculus where we do not use any locality based feature. Another immediate consequence of receptiveness is:

COROLLARY 4.8. *If $a(\vec{u}).P$ is well-formed and typable, then $P \not\downarrow a$.*

This property says that a receiver is *persistent*, in the sense that it will still be available, under the same name, whatever messages are sent to it. Therefore we may call such a receiver a "server", that always accept requests (i.e. messages), or a "resource". We now turn to the issue of message delivery, showing that, if a message is sent (at some locality) on a channel of a known scope in a well-formed and typed process, then the process offers a receiver for this message.

THEOREM (MESSAGE DELIVERABILITY) 4.9. *Let S be a well-formed and typable term, with $I \Vdash S$. If $S \stackrel{*}{\rightarrow} (\nu\vec{w})([\ell :: \overline{a}(\vec{v})] \mid S_0)$ with $a@\ell \in I$ or $a \in \mathsf{subj}(\vec{v})$, then there exist R and S_1 such that $S_0 \stackrel{*}{\rightarrow\!\!\!\!\!\rightarrow} (\nu\overrightarrow{w'})([\ell :: a(\vec{u}).R] \mid S_1)$ with $a \notin \mathsf{subj}(\overrightarrow{w'})$.*

PROOF: by the Proposition 4.5, we have $I \Vdash (\nu\vec{w})([\ell :: \overline{a}(\vec{v})] \mid S_0)$ with $a@\ell \in I$ or $a \in \mathsf{subj}(\vec{w})$. We first show by induction on the length of $\vec{w}$ that this implies $S_0 \not\downarrow a@\ell$. If this length is 0 we have $I \Vdash S'$ and $a@\ell \in I$, and we conclude using the Proposition 4.7. Otherwise $\vec{w} = w, \overrightarrow{w_0}$. Let us assume that $a = \mathsf{subj}(w)$ and $a \notin \mathsf{subj}(\overrightarrow{w_0})$ (otherwise $a@\ell \in I$ or $a \in \mathsf{subj}(\overrightarrow{w_0})$ and we simply use the induction hypothesis). Since S is typed, say with $\Gamma \vdash S$, we also have $\Gamma \vdash (\nu\vec{w})([\ell :: \overline{a}(\vec{v})] \mid S_0)$ by the Theorem 3.4. Now suppose that $w = a : \tau$; then we would have

$$\dfrac{\dfrac{\dfrac{\vdots}{a : \tau\,,\ \Delta \vdash (\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}{\Delta \vdash (\nu a : \tau)(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}{\dfrac{\vdots}{\Gamma \vdash (\nu a : \tau)(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}$$

with $\Delta \subseteq \Gamma$. Since a is free in $(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)$, it must be the case (see below) that Δ contains $\ell : \{a : \gamma\}$ for some γ, but then $a : \tau, \Delta$ is not a typing context. Therefore we have $w = a@\ell'$, and

$$\dfrac{\dfrac{\dfrac{\vdots}{\ell' : \{a : \gamma\}\,,\ \Delta \vdash (\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}{\Delta \vdash (\nu a@\ell')(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}{\dfrac{\vdots}{\Gamma \vdash (\nu a@\ell')(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}$$

with $\Delta \subseteq \Gamma$ and a not in Δ. Since a is free in $(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)$, and the typing of $[\ell :: \overline{a}(\vec{v})]$ has the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Theta \vdash_\ell \vec{v} : \vec{\tau}}}{\ell : \{a : Ch(\vec{\tau})\}\,,\ \Theta \vdash_\ell \overline{a}(\vec{v})}}{\dfrac{\dfrac{\vdots}{\Sigma \vdash_\ell \overline{a}(\vec{v})}}{\Sigma \vdash [\ell :: \overline{a}(\vec{v})]}}}{\dfrac{\vdots}{\Delta' \vdash [\ell :: \overline{a}(\vec{v})]}}$$

with $\ell : \{a : Ch(\vec{\tau})\}, \Theta \subseteq \Sigma \subseteq \Delta'$, the context $\ell' : \{a : \gamma\}, \Delta$ must contain the assumption $\ell : \{a : Ch(\vec{\tau})\}$. Since a do not occur in Δ, we have $\ell' = \ell$, and $\gamma = Ch(\vec{\tau})$. Then we have

$$\dfrac{\dfrac{\dfrac{\vdots}{a@\ell\,,\ I \Vdash (\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}}{I \Vdash (\nu a@\ell)(\nu\overrightarrow{w_0})([\ell :: \overline{a}(\vec{v})] \mid S_0)}$$

therefore $S_0 \not\downarrow a@\ell$ by induction hypothesis.

Now by definition $S_0 \stackrel{*}{\twoheadrightarrow} S_1$ for some S_1 such that $S_1 \downarrow a@\ell$, that is $S_1 \equiv (\nu\overrightarrow{w_1})([\ell' :: P] \mid S_2)$ with $P \downarrow v$, $v@\ell' = a@\ell$ and $a, \ell \notin \mathsf{subj}(\overrightarrow{w_1})$. Then we have either $P \downarrow a$ and $\ell' = \ell$, or $P \downarrow a@\ell$. In the first case we have

$$S_1 \equiv (\nu\overrightarrow{w_1})([\ell :: (\nu\overrightarrow{w_2})(a(\vec{u}).R \mid Q)] \mid S_2) \equiv (\nu\overrightarrow{w_3})([\ell :: a(\vec{u}).R] \mid ([\ell :: Q] \mid S_2))$$

with $a \notin \mathsf{subj}(\overrightarrow{w_1}, \overrightarrow{w_2}, \overrightarrow{w_3})$, concluding the proof in that case. If $P \downarrow a@\ell$ then by the Lemma 4.6 there exists P' such that $P' \downarrow a$ and $[\ell' :: P] \stackrel{*}{\rightarrow} (\nu \overrightarrow{w_2})([\ell :: P'] \mid S_3)$ for some S_3, with $a \notin \mathsf{subj}(\overrightarrow{w_2})$, and we conclude as in the previous case. ⧉

Note that this result does not hold for untyped terms. For instance we have $\Vdash (\nu a : val)\overline{a}$ or $\Vdash (\nu a : \emptyset)\overline{a}$, and these terms contain a message that cannot be delivered. From this result one can show a similar property for the π_1^r-calculus, namely that if P is well-formed and typable, with $I \Vdash P$, and if $P \stackrel{*}{\rightarrow} (\nu \vec{c})(\overline{a}(\vec{v}) \mid R)$ with $a \in I \cup \{\vec{c}\}$ then there exist R' and Q such that $R \stackrel{*}{\rightarrow} (\nu \vec{b})(a(\vec{u}).R' \mid Q)$.

5. Receptive Distributed Programming.

In this section we give some examples to illustrate the style of "programming" that one must adopt to conform to the receptive discipline. One may wonder for instance how we could "program" typically non-receptive agents such as the "one-slot buffer" that repeatedly waits for some data on a given channel and then sends it on another channel. In the synchronous π-calculus, this process may be written:

$$\big(\mathsf{rec}\, B(a,b).a(c).\overline{b}(c).B(a,b)\big)(a,b)$$

Clearly, this cannot be written so easily in $\mathrm{D}\pi_1^r$. In the following examples we use the standard notation $\mathsf{if}\ a = b\ \mathsf{then}\ P\ \mathsf{else}\ Q$ – and more generally $\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ Q$ – for $[a = b]P, Q$. We denote by $\overline{a}(\vec{v}, \_, \vec{v'})$ the term $(\nu b)(\overline{a}(\vec{v}, b, \vec{v'}) \mid T_b)$, or $(\nu b : val)\overline{a}(\vec{v}, b, \vec{v'})$, depending on the type of a. We recall that we write a recursive process $(\mathsf{rec}\, A(\vec{u}).P)(\vec{u})$ that does not introduce new parameters simply as $\mathsf{rec}\, A(\vec{u}).P$.

Example 1: Buffers

We may program the one-slot buffer as follows: first it inputs on a a message that is supposed to convey a datum to store in the buffer (if this is not the case the message is ignored, i.e, it is resent), and then on the *same* channel it receives a request for extracting the contents of the buffer, which is delivered on a private return channel (again, if this protocol is violated this second message is ignored, though obviously something more elaborate could be done in a more synchronous version of the buffer).

$$\begin{array}{rl} \mathit{Buff}_1(a) = \mathsf{rec}\, B(a).a(k_1,x,y).\mathsf{if}\ k_1 \neq \mathsf{put} & \\ \quad \mathsf{then}\ \overline{a}(k_1,x,y) \mid B(a) & \\ \quad \mathsf{else}\ \mathsf{rec}\, B_1(a,x).a(k_2,z,y).\mathsf{if}\ k_2 \neq \mathsf{get} & \\ \qquad\qquad \mathsf{then}\ \overline{a}(k_2,z,y) \mid B_1(a,x) & \\ \qquad\qquad \mathsf{else}\ \overline{y}(x) \mid B(a) & \end{array}$$

The reader can check that $a \Vdash \mathit{Buff}_1(a)$, and that the typing judgement $\ell : \{a : Ch(val, \tau, Ch(\tau))\} \vdash_\ell \mathit{Buff}_1(a)$ is valid. The requests for reading and writing the buffer are respectively $(\nu c)(\overline{a}(\mathsf{get}, \_, c) \mid c(x){:}P)$ (where we may assume that the content of the buffer is of value type, i.e. $\tau = val$) and $\overline{a}(\mathsf{put}, b, \_)$. As one can see, the buffer is now a kind of "agent", that is a process which is invoked by its name a and reacts according to some internal protocol. To build a "two-slots buffer" from this one, we may proceed as usual, putting together two one-slot buffers with a private communication between them. However, to write this we need to refine our previous program, because we need to explicitly indicate the keys used, defining $\mathit{Buff}_1(a, \mathsf{put}, \mathsf{get})$ – in the obvious way. This is left as an exercise for the reader (easier in π^r than in π_1^r).

Example 2: Mutual Exclusion

Synchronization can be "programmed" in the π-calculus, a typical example being mutual exclusion between tasks enforced by the use of a *lock*, as follows:

$$\begin{array}{rl} (\nu l)\big(\overline{l} \mid & \mathsf{task}_1{}^*(\vec{u}).l().(\cdots \overline{l}) \\ & \vdots \\ \mid & \mathsf{task}_n{}^*(\vec{v}).l().(\cdots \overline{l})\big) \end{array}$$

This violates both the unique receiver property and receptiveness. In the receptive style, the lock is represented by a process that receives on a unique channel l messages carrying a value lock or unlock and a return channel r:

$$\begin{array}{rcl}\mathsf{Lock}(l) & = & \mathsf{rec}\,A(l).l(k,r).\mathsf{if}\ k \neq \mathsf{lock} \\ & & \quad \mathsf{then}\ \overline{l}(k,r) \mid A(l) \\ & & \quad \mathsf{else}\ \overline{r} \mid \mathsf{rec}\,A'(l).l(k',s).\mathsf{if}\ k' \neq \mathsf{unlock} \\ & & \qquad \mathsf{then}\ \overline{l}(k',s) \mid A'(l) \\ & & \qquad \mathsf{else}\ A(l)\end{array}$$

and a task is now written

$$\mathsf{task}_i{}^*(\vec{u}).(\nu r)\big(\,\overline{l}(\mathsf{lock},r)\ \mid\ r():(\,\cdots\ \overline{l}(\mathsf{unlock},\_))\big)$$

Again, the lock is now a persistent agent that has an identity l, and reacts according to its own protocol, governed by the keys it receives.

Example 3: An Object Server

Suppose we have written a generic "object" $obj(b,\vec{d})$, with name b – that is, $b \Vdash obj(b,\vec{d})$ – and state parameters $\vec{d}$, and that we would like to create at some location ℓ_0 a server for that kind of objects, which will deliver instances of it on requests to a name s – this is similar to *code on demand*. Then we write this as follows, assuming that the parameters $\vec{d}$ are values:

$$s^*(c@\ell,\vec{d}).\mathsf{go}\,\ell.(\nu b)(\overline{c}(b) \mid obj(b,\vec{d}))$$

A client at site ℓ_1 that wants to acquire its own instance of the object, and use it in the process P will be written:

$$(\nu c)(\,\overline{s}@\ell_0(c@\ell_1,\vec{d})\ \mid\ c(b):P\,)$$

We let the reader check that, if we can type $obj(b,\vec{d})$ with type ζ for b, then these terms are typable with $\ell_0 : \{s : Ch(Ch(\zeta)^{@},\overrightarrow{val})\}$.

Example 4: A Distributed Object

In this example we imagine an "object" which is located both at ℓ_0 and ℓ_1 (with the same name), and has a shared, possibly distributed state. More specifically, we program a distributed "button", named a. "Pushing the button" at ℓ_i means sending a message to the channel a at ℓ_i. Pushing twice the button at either ℓ_0 or ℓ_1 results in the emission of a message c at ℓ. In particular, pushing once at ℓ_0 and once at ℓ_1 will produce an emission. This condition obviously forces some cooperation between the two locations. We seek a solution that will allow emission even when one of the two locations stops working. This condition rules out centralized solutions where emissions are always decided at one location. On the other hand, some degree of asymmetry seems desirable to rule out a circular situation where each location tries to "borrow a push" from the other location in order to produce an emission. In the presented solution, the button at location ℓ_0 is allowed to borrow from the one at ℓ_1, but not the other way round. The distributed button relies on a private value b – to borrow a push:

$$S = (\nu b : val)\big([\ell_0 :: P_0] \mid [\ell_1 :: P_1]\big)$$

where

$$\begin{array}{rcl}P_0 & = & \mathsf{rec}\,A(a).a(i).\big(\overline{a}@\ell_1(b) \mid a(i).(A(a) \mid \overline{c}@\ell())\big) \\ P_1 & = & \mathsf{rec}\,A'(a).a(i).\mathsf{if}\ i = b\ \mathsf{then}\ (A'(a) \mid \overline{a}(b)) \\ & & \qquad \mathsf{else}\ a(i).\mathsf{if}\ i = b\ \mathsf{then}\ (A'(a) \mid \overline{a}@\ell_0(b)) \\ & & \qquad\quad \mathsf{else}\ (A'(a) \mid \overline{c}@\ell())\end{array}$$

It is easy to see that these terms are well-formed, with interface $\{a\}$, and therefore $a@\ell_0, a@\ell_1 \Vdash P$, and typable with $\ell_0,\ell_1 : \{a : Ch(val)\}$. The message $\overline{a}@\ell_1(b)$ is interpreted by P_1 as a request to borrow a push, while the message $\overline{a}@\ell_0(b)$ is interpreted by P_0 as a push lent by P_1. This solution could be refined. For instance, the processes may want to check that the request to borrow received by the process at ℓ_1 corresponds to the *current* request to borrow from the process at ℓ_0 (if it is not the case then the offer to lend is declined). Similar, but more complicated techniques could be applied for the programming of a distributed channel manager or, more generally, of a distributed memory.

Notice that to fix the scope of the name of such a distributed object, we should extend the syntax to allow $(\nu a@L)P$, where L is a non-empty finite set of (location) names. It is then straightforward to modify the type system and the system for well-formedness to deal with this extension.

Example 5: Migrating Cells

In this example, we shall adopt a notation similar to the one for "objects" of TyCO [22], that is we will write

$$a\{k_1 = (\vec{u}_1)P_1, \ldots, k_n = (\vec{u}_n)P_n, Q\}$$

for the process

$$\begin{aligned} a(k, \vec{u}_1, \ldots, \vec{u}_n).\, &[k = k_1]\, P_1, \\ &\vdots \\ &[k = k_n]\, P_n, Q \end{aligned}$$

where the k_i are assumed to be of type *val*. This is well-formed if $a \Vdash P_i$ for all i, and also $a \Vdash Q$. Such an agent behaves as an "object", or more accurately as an "actor" of name a, offering "methods" $k_1, \ldots, k_n$, and possibly changing its behaviour – but not its identity – after having received a message. The messages for such an agent may be written $a \triangleleft k_i(\vec{u}_i)$, which abbreviates $\overline{a}(k_i, \_, \ldots, \vec{u}_i, \ldots, \_)$. Admittedly, in a more language oriented version of the Dπ_1^τ-calculus it would be preferable to adopt as primitive the notations (and typing) of TyCO; we just use them for the sake of illustration here. In many cases these objects have a simple recursive behaviour, given as follows:

$$\begin{aligned} \mathsf{rec}\, A(a, \vec{v}).a(k, \vec{u}_1, \ldots, \vec{u}_n).\, &[k = k_1]\, (P_1 \mid A(a, \vec{v})), \\ &\vdots \\ &[k = k_n]\, (P_n \mid A(a, \vec{v})), A(a, \vec{v}) \end{aligned}$$

where $\Vdash P_i$ for all i. Such an agent reacts uniformly to the "method calls", by performing the corresponding process P_i and returning itself.

In the same style as for the buffer, we could program a *cell* (see [21]), which we may read without destroying its content. To make this example more interesting, we suppose that the cell not only responds to requests for reading and writing, but also to an instruction to *migrate* to some given location. The name c of the cell has type $\gamma = Ch(val, Ch(val), val, Ch(), loc)$, where the first argument is the key for reading, writing and migrating, the second and fourth arguments are return channels, for readers and writers respectively, the third argument is the content of the cell, and the last, of locality type $loc = \{\}$, is the destination locality in case of migration. To maintain some consistency, the cell moved at a new location bears a new name, which is used to forward messages arriving at the location it originates from. Notice that if we keep the same name our example is not well-formed. The migrating cell $MigCell(c, x, \ell)$ has parameters x and ℓ, which are its current value and location; it is the following term:

$$\begin{aligned} MigCell(c, x, \ell) = \mathsf{rec}\, A(c, x, \ell).c\{\, &\mathsf{read} = (y)(\overline{y}(x) \mid A(c, x, \ell))\,, \\ &\mathsf{write} = (x', z)(\overline{z}() \mid A(c, x', \ell))\,, \\ &\mathsf{migr} = M\,,\ A(c, x, \ell)\} \end{aligned}$$

where

$$\begin{aligned} M &= (\ell')(\nu c'@\ell')\big(\mathsf{go}\,\ell'.A(c', x, \ell') \mid Proxy\big) \\ Proxy &= c^*(k, y, x, z, t).\mathsf{go}\,\ell'.(\nu y')\big(\, Fwd(y', y@\ell) \;\mid \\ &\qquad (\nu z')\big(\, Fwd(z', z@\ell) \;\mid \\ &\qquad\quad \overline{c'}(k, y', x, z', t)\,\big)\big) \end{aligned}$$

where, as we have seen, the "forwarder" is

$$Fwd(a, b@\ell) = a^*(\vec{x}).\overline{b}@\ell(\vec{x})$$

In case a read operation, that is $\overline{c} \triangleleft \mathsf{read}(y)$, has to be processed, the cell returns its current value x on the channel y to the reader. In case of a write operation $\overline{c} \triangleleft \mathsf{write}(x', z)$, the cell is updated with the new value x', and an acknowledgement z is sent back to the writer. Notice that readers and writers are always supposed to be local processes, or more precisely processes at the same locality as the cell. Finally a migration message $\overline{c} \triangleleft \mathsf{migr}(\ell')$ causes the cell to be *moved* at the destination locality ℓ', with a new name c', its current content x and a new current location ℓ', while a "proxy" managing the calls to c is set up at the locality that the cell is leaving.

The reader can check that $MigCell(c, x, \ell)$ is both well-formed, with interface $\{c\}$, and typable, with c of type γ at the current location ℓ, and x of type val. In this version, the remote cell can only be accessed by messages emitted in the location it originated from. We could also make the new cell available at its new location, on a name a received as argument, that is changing M into:

$$M = (a@\ell')(\nu c'@\ell')\big(\mathsf{go}\,\ell'.(\overline{a}(c') \mid A(c', x, \ell')) \mid Proxy\big)$$

Then a process using this cell and wishing to migrate together with this part of its state would typically execute:

$$Q = (\nu a@\ell')\big(\overline{c} \triangleleft \mathsf{migr}(a@\ell') \mid \mathsf{go}\,\ell'.a(c'){:}P\big)$$

It is easy to see that we have the following behaviour:

$$[\ell :: MigCell(c, x, \ell) \mid Q \mid R] \xrightarrow{*} (\nu c'@\ell')\big([\ell :: Proxy \mid R] \mid [\ell' :: MigCell(c', x, \ell') \mid P \mid (\nu a)\,T_a]\big)$$

However, to type this new version, we need recursive types. These are introduced by adding type variables and recursion, as follows:

$$\tau ::= \cdots \mid t \mid \mu\, t.\tau$$

These are used in the type system as usual, that is, identifying a recursive type with the (infinite, regular) tree it determines (notice that there is also an implicit notion of equality of location types, that may be regarded as record types). Now if we let

$$\delta = \mu\, t.Ch(val, Ch(val), val, Ch(), Ch(t)^{@})$$

we can type the new migrating cell with c of type δ.

Our migrating cell – say, the second version, with recursive types –, is only one possible kind of "state mobility". In [19] Sekiguchi and Yonezawa identify several other "mobility types". Our previous example is of what they call "carry type", where a migrating user of the cell carries it with him, while leaving access to the cell by means of a proxy. The "copy type", where the cell is duplicated, with no relation between the two copies, is easily written, by just changing the code of the migration method, as follows:

$$M_{copy} = (a@\ell')\big(\mathsf{go}\,\ell'.(\nu c')(\overline{a}(c') \mid A(c', x, \ell')) \mid A(c, x, \ell)\big)$$

Similarly, a migrating cell of "resident type" – the "dual" of "carry type" –, which stays in its location but may be accessed remotely via a proxy, is written:

$$M_{resident} = (a@\ell')\big(\mathsf{go}\,\ell'.(\nu c')(\overline{a}(c') \mid Proxy') \mid A(c, x, \ell)\big)$$

where $Proxy'$ is the "dual" of $Proxy$, accepting requests on c' at ℓ' and sending them back to c at ℓ. There are two other mobility types, called "proper" and "takeaway" in [19], where the link of the migrating (resp. immobile) user with the cell is lost, that do not seem to be expressible in our calculus, where any request may be received. However, we may use the "fake receiver" T_a to simulate the corresponding migrating cells:

$$\begin{aligned} M_{proper} &= (a@\ell')\big(\mathsf{go}\,\ell'.(\nu c')(\overline{a}(c') \mid T_{c'}) \mid A(c, x, \ell)\big) \\ M_{takeaway} &= (a@\ell')\big(\mathsf{go}\,\ell'.(\nu c')(\overline{a}(c') \mid A(c', x, \ell')) \mid T_c\big) \end{aligned}$$

In this particular example, we see that the discipline of receptive programming imposes that we cannot move a resource without providing the locality it is leaving with a replacement behaviour. Then we cannot actually (re)move a resource: what we may do is dynamically change its behaviour, and create new remote resources.

This example suggests that we could use our distributed π-calculus as a low-level "assembly" language, in which to encode higher-level migrating agents that move with part of their state. We could also use our calculus to encode more abstract migration schemes, like for instance "transparent routing", which is assumed in the JOIN-calculus [10]. More specifically, we may assume that we have a set of "global names", whose locations are known to the implementation by means of a mapping $\mathcal{L}$. Then we would like to program with these names without having to explicitly move the corresponding messages to their destination, writing simply $\overline{a}(\vec{u})$ where we actually mean $\overline{a}@?(\vec{u})$. Then the "implementation" of this transparent routing should be clear: given the mapping $\mathcal{L}$ assigning localities to "global names", we would translate an input as

$[\![a(\vec{b}).P]\!] = a(\overrightarrow{b@\ell_b})[\![P]\!]\mathcal{L}[\overrightarrow{b \mapsto \ell_b}]$ – assuming that all the received names are global ones –, and similarly for the output $[\![\overline{a}(\vec{b})]\!] = a@\mathcal{L}(a)(\overrightarrow{b@\mathcal{L}(b)})$.

6. Encoding the π_1-Calculus.

In this section, we show that there is a translation from the π-calculus to π_1^r. It is shown in [3] that the joined input of the join-calculus [9] can be defined in the π_1-calculus up to weak asynchronous bisimulation. On the other hand, it has been shown in [9] that there is a fully abstract translation of the asynchronous π-calculus in the join-calculus. Therefore, if we can translate the π_1-calculus in the π_1^r-calculus, we can reasonably claim that the nice properties of the latter are not obtained at the expense of its expressive power. In this informal argument, we are making the reasonable assumption that the translation from the asynchronous π-calculus to the join-calculus can be extended to include a conditional operator on *values*. We also note that embedding of the join-calculus into the π_1-calculus, we do not need the conditional operator on values, whereas the translation from the π_1-calculus to the π_1^r-calculus will make an essential use of it.

Let us first recall the definition of the π_1-calculus [2, 3], in a form that makes it a super calculus of π_1^r. In this calculus we only use simple names a, b, $c\ldots$, and no migrating construct, but we will use parametric processes with *arity*. More precisely, parametric process identifiers are now pairs made of an identifier A and a non-negative integer k, written A^k. That is, the syntax of π_1 is:

$$\begin{array}{rcl} w & ::= & a \mid a : val \\ P,\ Q,\ R\ldots & ::= & \mathbf{0} \mid \overline{a}(\vec{b}) \mid a(\vec{b}).P \mid (P \mid Q) \mid (\nu w)P \mid T(\vec{b}) \\ T & ::= & A^k \mid (\mathsf{rec}\, A^k(\vec{b}).P) \end{array}$$

subject to the same implicit requirements as for the distributed calculus of Section 2. Notice that A^k and A^n are considered different identifiers if $k \neq n$, and substitution $[T/A^k]P$ is such that

$$[T/A^k]B^n = \begin{cases} T & \text{if } A^k = B^n \\ B^n & \text{otherwise} \end{cases}$$

It is understood that in the relation of α-conversion we only replace bound identifiers by fresh identifiers with the same arity. Syntactically, π_1^r is the *monadic* subset of π_1, where we only use monadic identifiers A^1. Reduction is as before –notice that the law of unfolding is now formulated:

$$(\mathsf{rec}\, A^k(\vec{b}).P)(\vec{c}) \rightarrow [(\mathsf{rec}\, A^k(\vec{b}).P)/A^k][\vec{c}/\vec{b}]P$$

The system for checking well-formedness in π_1 is essentially the same as for π_1^r, except for input, and parametric processes: we now use judgements $\Vdash T : k$ indicating their arity. The rules are modified as follows:

$$\frac{a\,,\, I \Vdash P}{a\,,\, I \Vdash a(\vec{b}).P}$$

$$\frac{}{\Vdash A^k : k} \qquad \frac{a_1, \ldots, a_k \Vdash P}{\Vdash (\mathsf{rec}\, A^k(a_1, \ldots, a_k, \vec{b}).P) : k} \qquad \frac{\Vdash T : k}{a_1, \ldots, a_k \Vdash T(a_1, \ldots, a_k, \vec{b})}$$

As one can see, the arity of a well-formed parametric process is the number of names it can carry as input channels. The modified input rule allows to accept non-receptive processes, even in the "monadic" fragment, like $a().b().(T_a \mid T_b)$. Therefore we shall refer here to judgements of the system for checking receptiveness in π_1^r as $I \Vdash^r P$, while we keep the notation $I \Vdash P$ for the more general judgements regarding π_1. It is easy to see that a well-formed process has only one interface, that is $I \Vdash P$ and $I' \Vdash P$ implies $I = I'$, and that the Lemma 4.2 still holds. The typing system does not need to be modified, except obviously for the fact that a typing context Γ now assign channel types to identifiers A^k.

To state the properties of our translation from π_1 to π_1^r, we will use the standard concept of a bisimulation. In order to define bisimulations, we need to introduce *labeled transitions* for the terms of our calculus, or more precisely for π_1. That is, processes perform *actions* and evolve to new processes in doing so, which is denoted $P \xrightarrow{\alpha} P'$. The set of actions is given by:

$$\alpha ::= \tau \mid a\vec{b} \mid (\nu\vec{w})\overline{a}\vec{b}$$

$$(out)\ \frac{}{\overline{a}(\vec{b}) \xrightarrow{\overline{a}(\vec{b})} \mathbf{0}} \qquad (in)\ \frac{}{a(\vec{b}).P \xrightarrow{a\vec{c}} [\vec{c}/\vec{b}]P}$$

$$(ext)\ \frac{P \xrightarrow{\alpha} P'}{(\nu w)P \xrightarrow{(\nu w)\alpha} P'}\ \mathsf{subj}(w) \neq \mathsf{subj}(\alpha) \qquad (\nu)\ \frac{P \xrightarrow{\alpha} P'}{(\nu w)P \xrightarrow{\alpha} (\nu w)P'}\ \mathsf{subj}(w) \notin \mathsf{nm}(\alpha)$$

$$(cm)\ \frac{P \xrightarrow{(\nu \vec{w})\overline{a}\vec{b}} P'\ ,\ Q \xrightarrow{a\vec{b}} Q'}{P \mid Q \xrightarrow{\tau} (\nu \vec{w})(P' \mid Q')}\ \mathsf{subj}(\vec{w}) \cap \mathsf{fn}(Q) = \emptyset \qquad (cm')\ (symmetric)$$

$$(cp)\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset \qquad (cp')\ (symmetric)$$

$$(m_t)\ \frac{P \xrightarrow{\alpha} P'}{[a = a]P, Q \xrightarrow{\alpha} P'} \qquad (m_f)\ \frac{Q \xrightarrow{\alpha} Q'}{[a = b]P, Q \xrightarrow{\alpha} Q'}\ a \neq b$$

$$(rec)\ \frac{[(\mathsf{rec}\, A^k(\vec{b}).P)/A^k][\vec{c}/\vec{b}]P \xrightarrow{\alpha} P'}{(\mathsf{rec}\, A^k(\vec{b}).P)(\vec{c}) \xrightarrow{\alpha} P'} \qquad (id)\ \frac{P =_\alpha P' \quad P' \xrightarrow{\alpha} Q' \quad Q' =_\alpha Q}{P \xrightarrow{\alpha} Q}$$

Figure 8: Labelled transition system

meaning respectively internal communication (there should be no confusion between τ as a type and τ as an action), input of (simple) names $\vec{b}$ on the name a, and output of names, some of them possibly being private. In the last two cases we call a the subject of α, denoted $\mathsf{subj}(\alpha)$. In $(\nu\vec{w})\overline{a}\vec{b}$, we assume that $a \notin \{\overrightarrow{\mathsf{subj}(w)}\} \subseteq \{\vec{b}\}$, and we denote by $\mathsf{nm}(\alpha)$ (resp. $\mathsf{fn}(\alpha)$, $\mathsf{bn}(\alpha)$) the set of names occuring (resp. free, bound) in α. The labelled transition system presented in Figure 8 is the usual one for the polyadic π-calculus with early binding.

We now recall the definition of *asynchronous bisimulation* (henceforth simply bisimulation) for the π_1-calculus (see [1, 3]).

DEFINITION (BISIMULATION) *A relation $\mathcal{R}$ on well-formed processes is a bisimulation if it is symmetric, and if $P\,\mathcal{R}\,Q$ implies:*

(i) *P and Q have the same interface, i.e. $I \Vdash P$ and $I \Vdash Q$ for some I,*

(ii) *if $P \xrightarrow{\tau} P'$ then $Q \xrightarrow{\tau} Q'$ for some Q' such that $P'\,\mathcal{R}\,Q'$,*

(iii) *if $P \xrightarrow{(\nu \vec{w})\overline{a}\vec{b}} P'$, $a \notin I$, and $\{\overrightarrow{\mathsf{subj}(w)}\} \cap \mathsf{fn}(Q) = \emptyset$, then $Q \xrightarrow{(\nu \vec{w})\overline{a}\vec{b}} Q'$ for some Q' such that $P'\,\mathcal{R}\,Q'$,*

(iv) *if $P \xrightarrow{a\vec{b}} P'$ then there exists Q' such that either $Q \xrightarrow{a\vec{b}} Q'$ and $P'\,\mathcal{R}\,Q'$, or $Q \xrightarrow{\tau} Q'$ and $P'\,\mathcal{R}\,(Q' \mid \overline{a}(\vec{b}))$.*

We remark that this definition differs from the one for the asynchronous π-calculus (with multiple receivers, cf. [1]) in that an output action on a channel a of a well-formed process $I \Vdash P$ is observable only if the receiver for a is not already defined in P (condition (iii)). This entails that processes with different interfaces can in general be distinguished simply by putting them in parallel with a message on a channel which belongs only to one interface. For this reason, we require that bisimilar processes have the same interface (condition (i)). We denote with $\sim$ the largest bisimulation. Regarding the relations between labelled transitions and reduction in π_1^τ, one can prove the following:

PROPOSITION 6.1.

(i) $P \equiv Q \ \Rightarrow\ P \sim Q$,

(ii) $P \twoheadrightarrow P' \ \Rightarrow\ P \sim P'$,

(iii) $P \to P' \ \Rightarrow\ P \xrightarrow{\tau} P'$ *or* $P \sim P'$,

(iv) $P \xrightarrow{a\vec{b}} P' \;\Leftrightarrow\; P \stackrel{*}{\to} (\nu\vec{w})(a(\vec{b}).R \mid Q)$ *and* $P' \equiv ([\vec{c}/\vec{b}]R \mid Q)$ *with* $a \notin \{\overrightarrow{\mathsf{subj}(w)}\}$,

(v) $P \xrightarrow{(\nu\vec{w})\overline{a}\vec{b}} P' \;\Leftrightarrow\; P \stackrel{*}{\to} (\nu\overrightarrow{w'})(\overline{a}(\vec{b}) \mid Q)$ *with* $P' \equiv (\nu\overrightarrow{w''})Q$ *where* $\vec{w}$ *is the part of* $\overrightarrow{w'}$ *that binds free names of* $\vec{b}$ *(and not* a*), while* $\overrightarrow{w''}$ *is the rest,*

(vi) $P \xrightarrow{\tau} P' \;\Rightarrow\; \exists Q.\ P \stackrel{*}{\to} Q \;\&\; Q \to P'$.

The notion of weak bisimulation is obtained by replacing everywhere transitions $\xrightarrow{\alpha}$ with weak transitions $\stackrel{\alpha}{\Longrightarrow}$ defined as usual, that is $\stackrel{\alpha}{\Longrightarrow} = (\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^*$ if $\alpha \neq \tau$, and $\stackrel{\tau}{\Longrightarrow} = (\xrightarrow{\tau})^*$. We denote with $\approx$ the largest weak bisimulation, and by $\mathcal{F}$ the related monotonic operator on binary relations which has $\approx$ as largest fixed point. We recall the following basic properties of (asynchronous) bisimulation. The proof given in [1] requires some substantial case analysis for the input transition.

PROPOSITION 6.2.

(i) *The relation* $\approx$ *is an equivalence relation.*

(ii) *If* $P \approx Q$ *then* $(P \mid \overline{a}(\vec{b})) \approx (Q \mid \overline{a}(\vec{b}))$.

We also have the standard fact that we get the same relation $\approx$ if in the definition of weak bisimulation we require that a strong transition is matched by a weak one. As an exercise, the reader may check that the relation

$$\mathcal{R} = \{\, (T_a, (P_1 \cdots \mid (P_n \mid Id_a) \cdots) \mid P_i \in E \,\}$$

is a bisimulation, where E is the set made of $\mathbf{0}$ and all the messages $\overline{a}(\vec{b})$ on channel a. As a consequence, $T_a \approx Id_a$. We also need a few notions of bisimulation up to techniques. To this end, we define the partial order $\leq_{\mathcal{F}}$ on relations as

$$\mathcal{R}_1 \leq_{\mathcal{F}} \mathcal{R}_2 \;\Leftrightarrow\; \mathcal{R}_1 \subseteq \mathcal{R}_2 \;\&\; \mathcal{R}_1 \subseteq \mathcal{F}(\mathcal{R}_2)\ .$$

Let H be an operator on binary relations that is monotonic with respect to $\leq_{\mathcal{F}}$ – we also say that H preserves $\leq_{\mathcal{F}}$ in this case ([2]). Then we say that $\mathcal{R}$ is a *weak bisimulation up to* H if $\mathcal{R} \subseteq \mathcal{F}(H(\mathcal{R}))$. The following result is proved in [17].

PROPOSITION 6.3.

(i) *The family of operators preserving* $\leq_{\mathcal{F}}$ *is closed under composition and union.*

(ii) *If an operator* H *on binary relations preserves* $\leq_{\mathcal{F}}$ *and* $\mathcal{R}$ *is a weak bisimulation up to* H *then* $\mathcal{R} \subseteq \approx$.

The proof of (i) is immediate and the proof of (ii) amounts to build a weak bisimulation, starting from $\mathcal{R}$ and iterating the operator $\lambda\mathcal{S}.(\mathcal{S} \cup H(\mathcal{S}))$.

In the translation from π_1 to π_1^r presented below, we rely on some abbreviations. We already used the notation $\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ Q$ for simple conditions C. This notation extends to the case where C is a boolean combination of name equalities $a = b$ and inequalities $a \neq b$ in the obvious way:

$$\begin{aligned}
\mathsf{if}\ \neg C\ \mathsf{then}\ P\ \mathsf{else}\ Q &= \mathsf{if}\ C\ \mathsf{then}\ Q\ \mathsf{else}\ P\\
\mathsf{if}\ C \vee C'\ \mathsf{then}\ P\ \mathsf{else}\ Q &= \mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{if}\ C'\ \mathsf{then}\ P\ \mathsf{else}\ Q\\
\mathsf{if}\ C \wedge C'\ \mathsf{then}\ P\ \mathsf{else}\ Q &= \mathsf{if}\ C\ \mathsf{then}\ (\mathsf{if}\ C'\ \mathsf{then}\ P\ \mathsf{else}\ Q)\ \mathsf{else}\ Q
\end{aligned}$$

In π_1^r (where we omit the arity superscript), we use a default channel notation "_" in various contexts:

(i) in an output, as we have seen, $\overline{a}(\vec{b}, \_, \vec{b'})$ stands for $(\nu c)(\overline{a}(\vec{b}, c, \vec{b'}) \mid T_c)$ or $(\nu c\colon val)\overline{a}(\vec{b}, c, \vec{b'})$, depending on the type of a,
(ii) in an input, $a(\vec{b}, \_, \vec{b'}).P$ denotes $a(\vec{b}, c, \vec{b'}).P$ where $c \notin \mathsf{fn}(P)$,
(iii) in a parametric process instantiation, $T(\_, \vec{b})$ stands for $(\nu a)T(a, \vec{b})$, which is well-formed if $\Vdash T$,
(iv) in a recursive definition, $(\mathsf{rec}\, A(\_, \vec{b}).P)$ denotes $(\mathsf{rec}\, A(a, \vec{b}).(T_a \mid P))$, which is well-formed if $\Vdash P$.

In the proofs we also use the notation $P \xrightarrow{(\nu\vec{c})\overline{a}\vec{b}, \_, \vec{b'}} P'$, which stands for $P \xrightarrow{(\nu\vec{c}, c:val)\overline{a}\vec{b}, c, \vec{b'}} P'$ where $c \notin \mathsf{fn}(P)$.

Now the idea of the translation is rather simple: we turn any message on a channel a into a request to a *channel manager* $CM(a)$ for a, sending the arguments of the message together with a key *out*. Symmetrically, we turn any input on a into a request to $CM(a)$, sending a key *in* and a private return channel to actually

([2]) In [17], an operator that preserves $\leq_{\mathcal{F}}$ is called "respectful".

receive something. The channel manager will filter the messages according to the keys, and act as appropriate. This is similar to what one would do in implementing communication for a language like the π-calculus – see for instance [7, 21] –, except that one would exploit elaborate data structures like "pools" or queues to manage the input and output requests in a more realistic way. However, there is an attack on this encoding which compromises abstraction: the environment can send a request for input to the channel manager. We then authenticate the requests for input by introducing a restricted key in_a for every channel manager which is known only by the process that can actually input on that channel.

Formally, for every name a we assume a fresh name in_a (that is, not in the names of the translated processes). The name in_a has sort val and it is used as the key of the channel a. We translate a term P of the π_1-calculus with interface I, where $I = \{a_1, \ldots, a_n\}$, into the following process

$$\langle\!\langle I, P \rangle\!\rangle = (\nu in_{a_1}) \cdots (\nu in_{a_n})(CM(a_1, in_{a_1}) \mid \cdots \mid CM(a_n, in_{a_n}) \mid [\![P]\!])$$

where $[\![P]\!]$ is defined below, which turns out to be also a well-formed process of the π_1^r-calculus with interface $\emptyset$. Sorts are translated as follows:

$$\begin{array}{rcl} [\![val]\!] & = & val \\ [\![Ch(\tau_1, \ldots, \tau_n)]\!] & = & Ch(val, [\![\tau_1]\!], \ldots, [\![\tau_n]\!], Ch([\![\tau_1]\!] \ldots, [\![\tau_n]\!]), val, val) \end{array}$$

where the first argument is the input/output key of the channel, then come the arguments of the message to be delivered, followed by the type of the return channel to which they are actually sent, and then we have two keys for internal choice. The following transition rules describe the behaviour of the channel manager where we assume $j_1 \neq in_a$:

$$(CM_\tau) \quad (CM(a, in_a) \mid \overline{a}(j_1, \vec{b}_1, r_1, c_1, c_1') \mid \overline{a}(in_a, \vec{b}_2, r_2, c_2, c_2')) \xrightarrow{\tau} (CM(a, in_a) \mid \overline{r_2}(\vec{b}_1))$$

$$(CM_{in}) \quad (CM(a, in_a) \mid \overline{a}(in_a, \vec{b}_2, r_2, c_2, c_2')) \xrightarrow{a(j_1, \vec{b}_1, r_1, c_1, c_1')} (CM(a, in_a) \mid \overline{r_2}(\vec{b}_1))$$

By this specification, the channel manager matches a request for input with a request for output (the latter can be provided by the environment). A reduction such as (CM_τ) could be directly implemented in a join-calculus enriched with a filter condition on the received messages. However, to simplify the proofs we will in a first step reason with the axiomatic specification above. That is, we extend the π_1^r with a new constant $CM(a, in_a)$ (with two free parameters), which behaves as prescribed and is such that $a \Vdash^r CM(a, in_a)$. Then we will see how to implement the channel manager in the π_1^r-calculus, up to weak bisimulation. In order to be able to use the specification of CM, we must add a transition rule that allows structural manipulations to be performed:

$$P \xrightarrow{\alpha} P' \;\&\; Q \equiv P \;\Rightarrow\; Q \xrightarrow{\alpha} P'$$

In the π_1^r this is harmless since $\equiv$ is a strong bisimulation (see below). The translation $[\![P]\!]$ of processes is as follows:

$$\begin{array}{rcl} [\![\mathbf{0}]\!] & = & \mathbf{0} \\ [\![\overline{a}\vec{b}]\!] & = & \overline{a}(\_, \vec{b}, \_, \_, \_) \\ [\![a(\vec{b}).P]\!] & = & (\nu r)(\overline{a}(in_a, \_, r, \_, \_) \mid r(\vec{b}){:}[\![P]\!]) \\ [\![P \mid Q]\!] & = & ([\![P]\!] \mid [\![Q]\!]) \\ [\![[a = b]P, Q]\!] & = & [a = b][\![P]\!], [\![Q]\!] \\ [\![(\nu a)P]\!] & = & (\nu a)(\nu in_a)(CM(a, in_a) \mid [\![P]\!]) \\ [\![(\nu a : val)P]\!] & = & (\nu a : val)[\![P]\!] \end{array}$$

Regarding the translation of parametric processes, we assume given an injection that maps identifiers with arity A^k of π_1 into identifiers of π_1^r. For simplicity, we denote the image of A^k by A_k. Moreover, we assume in the following clauses that T has arity k, that is T is A^k or $(\mathsf{rec}\, A^k(\vec{b}).P)$, and that $\vec{b} = \vec{a}, \vec{c}$ where $\vec{a}$ is of length k if the length of $\vec{b}$ is greater than k, and $\vec{a} = \vec{b}$ (and $\vec{c}$ is empty) otherwise.

$$\begin{array}{rcl} [\![T(\vec{b})]\!] & = & [\![T]\!](\_, \vec{a}, \overrightarrow{in_a}, \vec{c}) \\ [\![A^k]\!] & = & A_k \\ [\![(\mathsf{rec}\, A^k(\vec{b}).P)]\!] & = & (\mathsf{rec}\, A_k(\_, \vec{a}, \overrightarrow{in_a}, \vec{c}).[\![P]\!]) \end{array}$$

We note that the only receivers in the translation are the channel managers and the "input once" return channels r in $r(\vec{b})$:$[\![P]\!]$. As expected, the translation preserves well-formedness:

PROPOSITION 6.4. *If $I \Vdash P$ in the π_1-calculus then $\Vdash^r [\![P]\!]$ and $I \Vdash^r \langle\!\langle I, P \rangle\!\rangle$.*

PROOF: by induction on the definition of $I \Vdash P$. ▭

LEMMA 6.5. *Suppose $I \Vdash P$ and let S be a substitution such that $[\mathsf{S}]P$ is defined, and S is injective on I. Then $[\mathsf{S}][\![P]\!] = [\![[\mathsf{S}]P]\!]$ and $[\mathsf{S}]\langle\!\langle I, P \rangle\!\rangle = \langle\!\langle \mathsf{S}(I), [\mathsf{S}]P \rangle\!\rangle$.*

Now we embark on the proof that our translation is fully abstract with respect to weak bisimulation. As a first step, let us define the relation $\cong$ on terms as the least relation satisfying the same clauses as structural equivalence $\equiv$ (restricted to π_1), plus:

$$\begin{array}{rcll} (\nu a : val)P & \cong & P & a \notin \mathsf{fn}(P) \\ (\nu a)T_a & \cong & \mathbf{0} & \end{array}$$

One can prove that this is a strong bisimulation, hence

LEMMA 6.6. $P \cong Q \;\Rightarrow\; P \sim Q$

We introduce a notion of deterministic reduction $>_d$ which will be used to handle communications on return channels, which are used only once. Namely, $P >_d P'$ if

$$P \cong \mathbf{E}[(\nu r)(\overline{r}(\vec{b}) \mid r(\vec{c}):Q)] \xrightarrow{\tau} \mathbf{E}[(\mathbf{0} \mid ([\vec{b}/\vec{c}]Q \mid Id_r))] \cong P' \cong \mathbf{E}[[\vec{b}/\vec{c}]Q]$$

where $r \notin \mathsf{fn}(Q)$ and $\mathbf{E}$ is any evaluation context. We also write $\geq_d$ and $<_d$, $\leq_d$ with obvious meanings. It is easily verified that $>_d$ commutes, modulo $\cong$, with any transition:

LEMMA 6.7. *If $P \xrightarrow{\alpha} P'$ and $P >_d P''$ then either $P' \cong P''$ or for some Q, $P' >_d Q$ and $P'' \xrightarrow{\alpha} Q$.*

We define the following operators on relations ([3]):

$$\begin{array}{rcl} E(\mathcal{R}) & = & \{\, ((P \mid M), (Q \mid M)) \mid (P, Q) \in \mathcal{R} \;\&\; M = \Pi_{i \in I} \overline{a_i}(\vec{b_i}) \,\} \\ H_1(\mathcal{R}) & = & \sim \circ E(\mathcal{R}) \circ \sim \\ H_2(\mathcal{R}) & = & \geq_d \circ \mathcal{R} \circ \leq_d \\ H_3(\mathcal{R}) & = & \{\, ((P \mid T_r), (Q \mid T_r)) \mid (P, Q) \in \mathcal{R} \,\} \end{array}$$

LEMMA 6.8.

(i) *The identity and the constant operators mapping a relation $\mathcal{R}$ to $\sim$, $\leq_d$, $\geq_d$, respectively, preserve $\leq_{\mathcal{F}}$.*

(ii) *The operators E and H_i $(i = 1, 2, 3)$ preserve $\leq_{\mathcal{F}}$.*

PROOF: (i) the identity obviously preserves $\leq_{\mathcal{F}}$. For $\lambda\mathcal{R}.\sim$ we observe that a strong bisimulation is a weak bisimulation. For $\lambda\mathcal{R}.\leq_d$ and $\lambda\mathcal{R}.\geq_d$ we note that $\leq_d$ is a weak bisimulation.
(ii) We note that all four operators are monotonic thus we just need to check that $\mathcal{R}_1 \leq_{\mathcal{F}} \mathcal{R}_2$ implies $H(\mathcal{R}_1) \subseteq \mathcal{F}(H(\mathcal{R}_2))$. ▭

The translation $\langle\!\langle I, P \rangle\!\rangle$ "simulates" P as follows:

LEMMA 6.9. *Suppose $I \Vdash P$. Then*

(i) *if $P \xrightarrow{a\vec{b}} P'$ then $\langle\!\langle I, P \rangle\!\rangle \xrightarrow{a(j,\vec{b},r,c,c')} Q >_d \langle\!\langle I, P' \rangle\!\rangle$,*

(ii) *if $P \xrightarrow{(\nu\vec{w})\overline{a}\vec{b}} P'$, $I' \Vdash P'$ and $a \notin I$ then $\langle\!\langle I, P \rangle\!\rangle \xrightarrow{(\nu\vec{w},r)\overline{a}(\_,\vec{b},r,\_,\_)} (\langle\!\langle I', P' \rangle\!\rangle \mid T_r)$ where $I' \Vdash P'$,*

(iii) *if $P \xrightarrow{\tau} P'$ then $\langle\!\langle I, P \rangle\!\rangle \xrightarrow{\tau} Q >_d \langle\!\langle I, P' \rangle\!\rangle$.*

PROOF SKETCH: we rely on the Lemma 6.5, and we proceed by induction on the definition of the labelled transition relation. ▭

In the output case (ii), we note the introduction of the T_r process acting on a fresh channel. We will show that we can factor out this spurious process. In particular, we observe the following property.

LEMMA 6.10. *Suppose $r \notin \mathsf{fn}(P \mid Q)$. Then $P \approx Q$ iff $(P \mid T_r) \approx (Q \mid T_r)$.*

([3]) Unlike the synchronous case, the operator $H_1'(S) = \sim \circ S \circ \sim$ does not respect $\leq_{\mathcal{F}}$. For instance, consider, in a language extended with sum, $a.\overline{a} + \tau \sim \tau\, S\, \tau$ with $S = \{(\tau, \tau), (\mathbf{0}, \mathbf{0})\}$. By computing first $E(S)$, we enforce closure under parallel composition with a message.

PROOF: ($\Rightarrow$) $\approx$ is preserved by parallel composition.
($\Leftarrow$) We show that the relation $\{\, (P,Q) \mid (P \mid T_r) \approx (Q \mid T_r) \ \&\ r \notin \mathsf{fn}(P \mid Q) \,\}$ is a weak bisimulation up to injective substitution. ❑

Since the key in_a is kept restricted, a message $\overline{a}(j, \vec{b}, r, c, c')$ received by a process $\langle\!| I, P |\!\rangle$ is always interpreted as an output request and therefore the fields j, r, c, c' are irrelevant. We formalize this remark as follows.

LEMMA 6.11. *Let $I \Vdash P$ and $a \in I$. Then for any j, r, c, c'*

$$\langle\!| I, (P \mid \overline{a}(\vec{b})) |\!\rangle \sim (\langle\!| I, P |\!\rangle \mid \overline{a}(j, \vec{b}, r, c, c'))$$

PROOF: let $M = \Pi_{i \in I} \overline{a_i}(j_i, \vec{b}_i, r_i, c_i, c'_i)$. We observe that

$$(CM(a, in_a) \mid \overline{a}(\_, \vec{b}, \_, \_, \_) \mid M) \sim (CM(a, in_a) \mid \overline{a}(j, \vec{b}, r, c, c') \mid M)$$

provided that $j \neq in_a$. ❑

Regarding the transitions of the translation $\langle\!| I, P |\!\rangle$, we have the following properties.

LEMMA 6.12. *Suppose $I \Vdash P$. Then:*
(i) *if $\langle\!| I, P |\!\rangle \xrightarrow{\tau} Q$ then $P \xrightarrow{\tau} P'$ and $Q >_d \langle\!| I, P' |\!\rangle$,*

(ii) *if $\langle\!| I, P |\!\rangle$ performs an output transition on a channel $a \notin I$ then $\langle\!| I, P |\!\rangle \xrightarrow{(\nu \vec{w}, r)\overline{a}(\_, \vec{b}, r, \_, \_)} (Q \mid T_r)$ and $P \xrightarrow{(\nu \vec{w})\overline{a}\vec{b}} P'$ for some I' and P' such that $I' \Vdash P'$ and $Q \cong \langle\!| I', P' |\!\rangle$,*

(iii) *if $\langle\!| I, P |\!\rangle \xrightarrow{a(j, \vec{b}, r, c, c')} Q'$ then $P \xrightarrow{a\vec{b}} P'$ for some P' such that $Q' >_d \langle\!| I, P' |\!\rangle$.*

PROOF SKETCH:
(i) the only receivers ready to execute in $\langle\!| I, P |\!\rangle$ are the channel managers. Thus a τ transition corresponds to an application of the rule (CM_τ) synchronising an input and an output request.
(ii) the only output actions arising from $\langle\!| I, P |\!\rangle$ with subject $a \notin I$ are those produced by the translation $[\![\overline{a}(\vec{b})]\!]$ of a message.
(iii) this input transition corresponds to the execution of the rule (CM_{in}). ❑

Now we can prove the main result of this section, showing that our translation into the calculus with constants $CM(a, in_a)$ is fully abstract with respect to weak bisimilarity.

THEOREM 6.13. *Suppose $I \Vdash P_1$ and $I \Vdash P_2$ in the π_1-calculus. Then*

$$P_1 \approx P_2 \ \Leftrightarrow\ \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle$$

PROOF: ($\Rightarrow$) we define

$$\mathcal{S} = \{\, (\langle\!| I, P_1 |\!\rangle, \langle\!| I, P_2 |\!\rangle) \mid P_1 \approx P_2 \,\}$$

We show that $\mathcal{S}$ is a bisimulation up to $H_3 \circ H_2 \circ H_1$. It follows from Proposition 6.3 that $\mathcal{S} \subseteq \approx$. Thus:

$$P_1 \approx P_2 \ \Rightarrow\ \langle\!| I, P_1 |\!\rangle \,\mathcal{S}\, \langle\!| I, P_2 |\!\rangle \ \Rightarrow\ \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle$$

We will only consider one half of the bisimulation condition, the other half follows by a symmetric argument, noting that the operators H_i preserve symmetry.
(τ) Suppose $\langle\!| I, P_1 |\!\rangle \xrightarrow{\tau} Q_1$. Then:

$P_1 \xrightarrow{\tau} P'_1 \ \&\ Q_1 >_d \langle\!| I, P'_1 |\!\rangle$	by Lemma 6.12(i),
$P_2 \stackrel{\tau}{\Longrightarrow} P'_2 \ \&\ P'_1 \approx P'_2$	since $P_1 \approx P_2$,
$\langle\!| I, P_2 |\!\rangle \stackrel{\tau}{\Longrightarrow} \langle\!| I, P'_2 |\!\rangle$	by Lemma 6.9(i).

Thus $Q_1 \geq_d \langle\!| I, P'_1 |\!\rangle \,\mathcal{S}\, \langle\!| I, P'_2 |\!\rangle$, as $P'_1 \approx P'_2$.

(*out*) Suppose $\langle\!| I, P_1 |\!\rangle \xrightarrow{(\nu\vec{w},r)\overline{a}(\_,\vec{b},r,\_,\_)} (\langle\!| I', P_1' |\!\rangle \mid T_r)$, according to Lemma 6.12(ii). Then:

$$\begin{array}{ll} P_1 \xrightarrow{(\nu\vec{w})\overline{a}\vec{b}} P_1' & \text{and} \\ P_2 \xRightarrow{(\nu\vec{w})\overline{a}\vec{b}} P_2' \;\&\; P_1' \approx P_2' & \text{since } P_1 \approx P_2, \\ \langle\!| I, P_2 |\!\rangle \xRightarrow{(\nu\vec{w},r)\overline{a}(\_,\vec{b},r,\_,\_)} (\langle\!| I', P_2' |\!\rangle \mid T_r) & \text{by Lemma 6.9.} \end{array}$$

Thus $(\langle\!| I', P_1' |\!\rangle \mid T_r)\ H_3(\mathcal{S})\ (\langle\!| I', P_2' |\!\rangle \mid T_r)$.

(*in*) Suppose $\langle\!| I, P_1 |\!\rangle \xrightarrow{a(j,\vec{b},r,c,c')} Q_1$. Then

$$\begin{array}{ll} P_1 \xrightarrow{a\vec{b}} P_1' \;\&\; Q_1 >_d \langle\!| I, P_1' |\!\rangle & \text{by Lemma 6.12(iii). Then either} \\ P_2 \xRightarrow{a\vec{b}} P_2' \;\&\; P_1' \approx P_2' & \text{since } P_1 \approx P_2, \text{ hence} \\ \langle\!| I, P_2 |\!\rangle \xRightarrow{a(j,\vec{b},r,c,c')} \langle\!| I, P_2' |\!\rangle & \text{by Lemma 6.9, or} \\ P_2 \xRightarrow{\tau} P_2' \;\&\; P_1' \approx (P_2' \mid \overline{a}(\vec{b})) & \text{by } P_1 \approx P_2, \\ \langle\!| I, P_2 |\!\rangle \xRightarrow{\tau} \langle\!| I, P_2' |\!\rangle & \text{by Lemma 6.9(i).} \end{array}$$

By Lemma 6.11, $\langle\!| I, P_2' \mid \overline{a}(\vec{b}) |\!\rangle \sim (\langle\!| I, P_2' |\!\rangle \mid \overline{a}(j,\vec{b},r,c,c'))$. Thus:

$$Q_1 >_d \langle\!| I, P_1' |\!\rangle\, \mathcal{S}\, \langle\!| I, P_2' \mid \overline{a}(\vec{b}) |\!\rangle \sim (\langle\!| I, P_2' |\!\rangle \mid \overline{a}(j,\vec{b},r,c,c'))$$

($\Leftarrow$) We define

$$\mathcal{S} = \{\, (P_1, P_2) \mid I \Vdash P_1,\ I \Vdash P_2 \;\&\; \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle \,\}$$

We show that $\mathcal{S}$ is a bisimulation. Thus

$$\langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle \;\Rightarrow\; P_1\, \mathcal{S}\, P_2 \;\Rightarrow\; P_1 \approx P_2\ .$$

(τ) Suppose $P_1 \xrightarrow{\tau} P_1'$. Then

$$\begin{array}{ll} \langle\!| I, P_1 |\!\rangle \xrightarrow{\tau} >_d \langle\!| I, P_1' |\!\rangle & \text{by Lemma 6.9(i),} \\ \langle\!| I, P_2 |\!\rangle \xRightarrow{\tau} Q_2 \;\&\; Q_2 \approx \langle\!| I, P_1' |\!\rangle & \text{since } \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle \\ P_2 \xrightarrow{\tau} P_2' \;\&\; Q_2 (>_d)^* \langle\!| I, P_2' |\!\rangle & \text{by Lemmas 6.12(i) \& 6.7.} \end{array}$$

Thus $\langle\!| I, P_1' |\!\rangle \approx Q_2 \geq_d \langle\!| I, P_2' |\!\rangle$ which implies $P_1'\, \mathcal{S}\, P_2'$.

(*out*) Suppose $P_1 \xrightarrow{(\nu\vec{w})\overline{a}\vec{b}} P_1'$ with $a \notin I$. Then:

$$\begin{array}{ll} \langle\!| I, P_1 |\!\rangle \xrightarrow{(\nu\vec{w},r)\overline{a}(\_,\vec{b},r,\_,\_)} (\langle\!| I', P_1' |\!\rangle \mid T_r) & \text{by Lemma 6.9(ii),} \\ \langle\!| I, P_2 |\!\rangle \xRightarrow{(\nu\vec{w},r)\overline{a}(\_,\vec{b},r,\_,\_)} Q_2 \;\&\; (\langle\!| I', P_1' |\!\rangle \mid T_r) \approx Q_2 & \\ P_2 \xrightarrow{(\nu\vec{w})\overline{a}\vec{b}} P_2' \;\&\; Q_2 (>_d)^* (\langle\!| I', P_2' |\!\rangle \mid T_r) & \text{by Lemmas 6.12, 6.7,} \\ & \text{and diagram chasing.} \end{array}$$

Thus $(\langle\!| I', P_1' |\!\rangle \mid T_r) \approx Q_2 (>_d)^* (\langle\!| I', P_2' |\!\rangle \mid T_r)$. By Lemma 6.10, $\langle\!| I', P_1' |\!\rangle \approx \langle\!| I', P_2' |\!\rangle$, and therefore $P_1'\, \mathcal{S}\, P_2'$.

(*in*) Suppose $P_1 \xrightarrow{a\vec{b}} P_1'$. Then $\langle\!| I, P_1 |\!\rangle \xrightarrow{a(j,\vec{b},r,c,c')} Q_1 >_d \langle\!| I, P_1' |\!\rangle$ by Lemma 6.9(i). There are two cases: if

$$\begin{array}{ll} \langle\!| I, P_2 |\!\rangle \xRightarrow{a(j,\vec{b},r,c,c')} Q_2 \;\&\; Q_1 \approx Q_2 & \text{by } \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle, \text{then} \\ P_2 \xRightarrow{a\vec{b}} P_2' \;\&\; Q_2 (>_d)^* \langle\!| I, P_2' |\!\rangle & \text{by Lemmas 6.12 \& 6.7.} \end{array}$$

Thus from $\langle\!| I, P_1' |\!\rangle \approx Q_1 \approx Q_2 \approx \langle\!| I, P_2' |\!\rangle$, it follows $P_1'\, \mathcal{S}\, P_2'$. In the other case, if

$$\begin{array}{ll} \langle\!| I, P_2 |\!\rangle \xRightarrow{\tau} Q_2 \;\&\; Q_1 \approx (Q_2 \mid \overline{a}(j,\vec{b},r,c,c')) & \text{by } \langle\!| I, P_1 |\!\rangle \approx \langle\!| I, P_2 |\!\rangle, \text{then} \\ P_2 \xRightarrow{\tau} P_2' \;\&\; Q_2 (>_d)^* \langle\!| I, P_2' |\!\rangle & \text{by Lemmas 6.12 \& 6.7.} \end{array}$$

$$
\begin{array}{rcl}
CM_1(a, in_a) & = & a(m_1).CM_2(a, m_1) \\
CM_2(a, m_1) & = & a(m_2).\mathsf{if}\ (j_1 = in_a \vee j_2 \neq in_a)\ \mathsf{then}\ (CM_1(a, in_a) \mid \overline{a}(m_1) \mid \overline{a}(m_2)) \\
 & & \qquad \mathsf{else}\ (\nu c)(\overline{a}(\_, \_, \_, c, c) \mid (\nu c')\overline{a}(\_, \_, \_, c, c') \mid CM_3(a, m_1, m_2, c)) \\
CM_3(a, m_1, m_2, c) & = & a(m_3).\mathsf{if}\ c_3 \neq c\ \mathsf{then}\ (CM_3(a, m_1, m_2, c) \mid \overline{a}(m_3)) \\
 & & \qquad \mathsf{else\ if}\ c_3' = c\ \mathsf{then}\ CM_4(a, m_1, m_2, c) \\
 & & \qquad\quad \mathsf{else}\ CM_5(a, m_1, m_2, c) \\
CM_4(a, m_1, m_2, c) & = & a(m_4).\mathsf{if}\ c_4 \neq c\ \mathsf{then}\ (CM_4(a, m_1, m_2, c) \mid \overline{a}(m_4)) \\
 & & \qquad \mathsf{else}\ (CM_1(a, in_a) \mid \overline{a}(m_1) \mid \overline{a}(m_2)) \\
CM_5(a, m_1, m_2, c) & = & a(m_5).\mathsf{if}\ c_5 \neq c\ \mathsf{then}\ (CM_5(a, m_1, m_2, c) \mid \overline{a}(m_5)) \\
 & & \qquad \mathsf{else}\ (CM_1(a, in_a) \mid \overline{r_2}(\vec{b}_1))
\end{array}
$$

Figure 9: Implementation of the channel manager

By Lemma 6.11, $\langle\!| I, P_2' \mid \overline{a}\vec{b} |\!\rangle \approx (\langle\!| I, P_2' |\!\rangle \mid \overline{a}(j, \vec{b}, r, c, c'))$. Thus from

$$\langle\!| I, P_1' |\!\rangle \approx Q_1 \approx (Q_2 \mid \overline{a}(j, \vec{b}, r, c, c')) \approx (\langle\!| I, P_2' |\!\rangle \mid \overline{a}(j, \vec{b}, r, c, c')) \approx \langle\!| I, P_2' \mid \overline{a}(\vec{b}) |\!\rangle$$

it follows $P_1' \,\mathcal{S}\, (P_2' \mid \overline{a}(\vec{b}))$. ❑

To conclude the proof of our result, it remains to show that the channel manager $CM(a, in_a)$ can be implemented adequately in π_1^r. In Figure 9 we show such an implementation. To denote concisely the various points of the control, it is convenient to describe it as a system of recursive equations where m_k, for $k = 1, \ldots, 5$, stands for the vector $j_k, \vec{b}_k, r_k, c_k, c_k'$ which is assumed not to contain in_a. It is immediate to compile this system of recursive equations in our notation $(\mathsf{rec}\, A(\vec{b}).P)$, that is, CM_1 really denotes the parametric recursive process

$$(\mathsf{rec}\, A_1(a, in_a).a(m_1).\underbrace{a(m_2).\mathsf{if}\ \cdots}_{CM_2(a, m_1)})$$

and similarly for $CM_i(a, m_1, m_2, c)$ (which are subterms of CM_1). One immediately verifies that $a \Vdash CM_1(a, in_a)$, as expected.

Let us comment on this implementation. The channel manager first performs two inputs on a. The first input must be an output request and the second an input request (otherwise the process CM_1 loops back). Then the channel manager proceeds to make an internal choice. To this end, it generates two messages $\overline{a}(\_, \_, \_, c, c)$ and $\overline{a}(\_, \_, \_, c, c')$, of which one may be received by CM_3, and then either by CM_4 or CM_5. If the message received in CM_3 is $\overline{a}(\_, \_, \_, c, c)$ the channel manager goes to state CM_4 and then loops back, otherwise it goes to state CM_5 and it enables a communication.

Let $[CM_1/CM]$ denotes the operation of replacing the abstract channel manager CM by the implementation CM_1 described above (if CM were regarded as an identifier, this would just be the substitution $[CM_1/CM]$). Then for instance $[CM_1/CM][\![P]\!]$ and $[CM_1/CM]\langle\!| I, P |\!\rangle$ are now terms of the π_1^r. For the following proposition, we still consider terms of the calculus enriched with the constants $CM(a, in_a)$.

PROPOSITION 6.14.

(i) *Let* $M \cong \Pi_{i \in I} \overline{a}_i(j_i, \vec{b}_i, r_i, c_i, c_i')$. *Then* $(CM(a, in_a) \mid M) \approx (CM_1(a, in_a) \mid M)$.

(ii) *If* $I \Vdash P$ *then* $\langle\!| I, P |\!\rangle \approx [CM_1/CM]\langle\!| I, P |\!\rangle$.

PROOF SKETCH:

(i) In this proof we use, in addition to $m_i = j_k, \vec{b}_k, r_k, c_k, c_k'$ as in the definition of CM_i (except that now we do not require that this vector does not contain in_a), the following notations:

$$
\begin{array}{rclcrcl}
M & \cong & \Pi_{i \in I} \overline{a}(j_i, \vec{b}_i, r_i, c_i, c_i') & \qquad & R & \cong & \Pi_{j \in J} \overline{r_j}(\vec{b}_j) \\
N & \cong & (M \mid R) & & m_{c_0, c_1} & = & \overline{a}(\_, \_, \_, c_0, c_1)
\end{array}
$$

Let $\mathcal{R}$ be the relation consisting of the following pairs:

$$\big((CM(a,in_a) \mid N)\,,\,(CM_1(a,in_a) \mid N)\big) \tag{1}$$

$$\big((CM(a,in_a) \mid \overline{a}(m_1) \mid N)\,,\,(CM_2(a,m_1) \mid N)\big) \tag{2}$$

$$\big((CM(a,in_a) \mid \overline{a}(m_1) \mid \overline{a}(m_2) \mid N)\,,\,((\nu c,c')(CM_3(a,m_1,m_2) \mid m_{c,c} \mid m_{c,c'}) \mid N)\big) \quad (*) \tag{3}$$

$$\big((CM(a,in_a) \mid \overline{a}(m_1) \mid \overline{a}(m_2) \mid N)\,,\,((\nu c,c')(CM_4(a,m_1,m_2) \mid m_{c,c'}) \mid N)\big) \quad (*) \tag{4}$$

$$\big((CM(a,in_a) \mid \overline{r}_2\vec{b}_1 \mid N)\,,\,((\nu c,c')(CM_5(a,m_1,m_2) \mid m_{c,c}) \mid N)\big) \tag{5}$$

$(*)$ where $j_1 \neq in_a$ and $j_2 = in_a$. Then one checks that $\mathcal{R}$ is a weak bisimulation. Let $(P,Q) \in \mathcal{R}$ and $P \xrightarrow{\alpha} P'$. For each of the cases (1-5), we argue that we can find a matching transition $Q \stackrel{\alpha'}{\Longrightarrow} Q'$ and still fall in one of the cases (1-5). In the following, we will describe schematically the matching transition by Q, omitting in particular the parameters of the CM_i's. For instance,

$$CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3$$

means that Q performs an input action on a, with appropriate arguments, preceeded and followed by internal synchronizations, and that following these transitions, the channel manager will move from state 4 to state 3 going through states 1 and 2. Whenever the sequence terminates in the state i, the reader should be able to verify that we fall in the i^{th} schema in the definition of the relation $\mathcal{R}$.

• If α is an output transition then it is caused by a message in R (its subject cannot be a, since a is in the interface of P). In all cases (1-5), the same message entails a matching transition by Q.

• Let us assume P's transition is obtained by means of rule CM_τ. We examine the five possible cases.

(1) To fire the transition, M must contain $\overline{a}(j_1,\vec{b}_1,r_1,c_1,c_1')$ and $\overline{a}(in_a,\vec{b}_2,r_2,c_2,c_2')$ with $j_1 \neq in_a$. Then Q matches P's transition with

$$CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(2) M must contain at least $\overline{a}(j_2,\vec{b}_2,r_2,c_2,c_2')$. Q matches with

$$CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(3) Q matches with

$$CM_3 \xrightarrow{\tau} CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(4) Q matches with

$$CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(5) Q matches with

$$CM_5 \xrightarrow{\tau} CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

• Next let us assume P's transition is obtained by means of rule CM_{in}.

(1) M must contain an input request $\overline{a}(in_a,\vec{b}_2,r_2,c_2,c_2')$ and $j_1 \neq in_a$. Q matches with

$$CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(2) We distinguish two cases.

(2.1) The transition consumes a message $\overline{a}(in_a,j,\vec{b},r,c,c') \in M$. Q matches with

$$CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(2.2) The transition consumes the message $\overline{a}(m_1)$ with $j_1 = in_a$. Q matches with

$$CM_2 \xrightarrow{a} CM_1 \xrightarrow{\tau} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(3) Q matches with

$$CM_3 \xrightarrow{\tau} CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(4) Q matches with

$$CM_4 \xrightarrow{\tau} CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

(5) Q matches with

$$CM_5 \xrightarrow{\tau} CM_1 \xrightarrow{a} CM_2 \xrightarrow{\tau} CM_3 \xrightarrow{\tau} CM_5 \xrightarrow{\tau} CM_1$$

• In the other direction, suppose $(P,Q) \in \mathcal{R}$ and $Q \xrightarrow{\alpha} Q'$. We proceed by a case analysis, as above.

(1) If CM_1 goes to CM_2 by an input or a synchronization then CM does nothing, i.e., $P \stackrel{\tau}{\Longrightarrow} P$ with 0 transitions, and we fall in case (2).

(2) If CM_2 goes to CM_3 by an input or a synchronization then CM does nothing and we fall in case (3). On the other hand, if CM_2 goes to CM_1 by an input or a synchronization then CM does nothing and we fall in case (1).

(3) If CM_3 loops on itself by an input or a synchronization then CM does nothing and we stay in case (3). On the other hand, if CM_3 goes to CM_4 by a synchronization then CM does nothing and we fall in case (4). Finally, if CM_3 goes to CM_5 by a synchronization then CM performs the corresponding synchronization and we fall in case (5).

(4) If CM_4 loops on itself by an input or a synchronization then CM does nothing and we stay in case (4). On the other hand, if CM_4 goes to CM_1 by a synchronization then CM does nothing and we fall in case (1).

(5) If CM_5 loops on itself by an input or a synchronization then CM does nothing and we stay in case (5). On the other hand, if CM_5 goes to CM_1 by a synchronization then CM does nothing and we fall in case (1).

(ii) In the translation, a channel manager $CM(a, in_a)$ can be in two positions:

(a) Under an input prefix, in a process of the shape:

$$(\nu a)(\nu in_a)(CM(a, in_a) \mid [\![P]\!])$$

We note that as long as $CM(a, in_a)$ (or $CM_1(a, in_a)$) is under the input prefix it does not play a role in the transitions and it is not affected by them because of the restrictions acting on its parameters.

(b) In a well-formed process that, up to structural equivalence, has the shape:

$$(\nu in_a)(CM(a, in_a) \mid P) \quad \text{or} \quad (\nu in_a)(\nu a)(CM(a, in_a) \mid P)$$

where P may contain the key in_a only in messages $\overline{a}(in_a, \vec{b}, r, c, c')$. Now in this case, one can show, by a case analysis similar to that in part (i), that the behaviour of $CM(a, in_a)$ placed in an evaluation context can be bisimulated by a suitable state $CM_i(a, in_a)$, $i = 1, \ldots, 5$ placed in the same evaluation context. ❑

7. Conclusion

In the area of sequential programming, one is usually not concerned with receptiveness, since the "resources" – what can be found in the heap or the memory – are assumed to be always available, although with a possibly mutable value. However this is only true in high-level languages, where no explicit manipulation of the memory is allowed. In low-level languages, like C, with a "free" instruction, one is potentially confronted with run-time errors where a program is trying to access to a non-existent resource. In this paper we have taken the view that, even for a low-level model like the π-calculus, it is a good discipline to "program" with persistent resources, which may always react in some way to requests – whether the reaction is the expected one is a matter of semantics, but the programmer of a service for instance is compelled to code some reaction in any state of this service.

We have shown that this discipline of receptive programming is, in the concurrent but non-distributed setting, actually not a limitation: with some effort, one can always encode non-receptive behaviours into receptive ones, while preserving weak bisimilarity. In the distributed and mobile setting, receptiveness, together with typing, ensures a highly desirable property, that we have called message deliverability – which

we could also be characterized as "no wandering message". We have shown, by means of examples, that the expressive power seems to remain very high in the distributed setting too. It would be very interesting to establish this as a formal result, for instance by encoding some language or calculus of migrating processes. This is left for further research.

We have not included this in the paper, but it should be obvious that there is an algorithm that, given a process, returns an interface if the process is well-formed (with the returned interface), and fails otherwise. Then, in order to show that receptiveness is indeed a sensible property, it remains to show that typing is feasible in some sense. One of the authors, C. Lhoussaine, is currently working on this problem; the aim is to show not only that typability is decidable, but also that given a typable process, one can infer a principal type, à la ML, for it, provided the names are syntactically partitioned among channels, locations, and values.

References

[1] R. Amadio, I. Castellani and D. Sangiorgi, *On bisimulations for the asynchronous π-calculus*, Theoretical Comput. Sci. Vol. 195 (1998) 291-324.

[2] R. Amadio, *An asynchronous model of locality, failure, and process mobility*, In Proc. COORDINATION'97, Springer Lect. Notes in Comp. Sci. 1282 (1997). Extended version appeared as Res. Report INRIA 3109.

[3] R. Amadio, *On modeling mobility*, Journal of Theoretical Computer Science Vol. 240, No. 1 (2000) 147-176.

[4] G. Berry and G. Boudol *The Chemical Abstract Machine* POPL'90 (1990) 81-94.

[5] G. Boudol, *Typing the use of resources in a concurrent calculus*, Proc. ASIAN 97, Springer Lect. Notes in Comp. Sci. 1345 (1997) 239-253.

[6] G. Boudol, I. Castellani, M. Hennessy and A. Kiehn, *Observing Localities*, Theoretical Computer Science 114 (1993) 31-61.

[7] L. Cardelli, *An implementation model of rendez-vous communication*, In Seminar on Concurrency, Springer Lect. Notes in Comp. Sci. 197 (1985) 449-457.

[8] L. Cardelli and A. Gordon, *Mobile ambients*, In Proc. FoSSaCS, ETAPS 98, Springer Lect. Notes in Comp. Sci. 1378 (1998) 140-155.

[9] C. Fournet and G. Gonthier, *The reflexive CHAM and the join-calculus*, In Proc. ACM Principles of Prog. Lang. (1996) 372-385.

[10] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy, *A calculus of mobile agents*, In Proc. CONCUR'96, Springer Lect. Notes in Comp. Sci. 1119 (1996) 406-421.

[11] M. Hennessy and J. Riely, *Resource access control in systems of mobile agents*, Techn. Report 2/98, School of Cognitive and Computer Sciences, University of Sussex (1998).

[12] N. Kobayashi, *A partially deadlock-free typed process calculus*, ACM ToPLaS Vol. 20 No 2 (1998) 436-482.

[13] N. Kobayashi, B. Pierce and D. Turner *Linearity and the π-calculus* ACM ToPLaS Vol. 21, No. 5 (1999) 914-947.

[14] M. Merro and D. Sangiorgi, *On asynchrony in name-passing calculi*, In Proc. ICALP'98, Springer Lect. Notes in Comp. Sci. 1443 (1998) 856-867.

[15] R. Milner, *The polyadic π-calculus: a tutorial*, Technical Report ECS-LFCS-91-180, University of Edinburgh (1991).

[16] D. Sangiorgi, *The name discipline of uniform receptiveness*, In Proc. ICALP'97, Springer Lect. Notes in Comp. Sci. 1256 (1997) 303-313.

[17] D. Sangiorgi, *On the bisimulation proof method*, Math. Struct. in Comp. Science Vol. 8, No. 5(1998) 447-479.

[18] D. Sangiorgi, *From λ to π; or, Rediscovering continuations*, Math. Struct. in Comp. Science Vol. 9, No. 4 (1999) 367-401.

[19] T. Sekiguchi and A. Yonezawa, *A calculus with code mobility* FMOODS'97, Chapman & Hall (1997).

[20] E. Sumii and N. Kobayashi, *A generalized deadlock-free process calculus*, In Proc. HLCL'98, Electronic Notes in Comp. Sci. vol. 16-3 (1998).

[21] D. Turner, *The Polymorphic Pi-Calculus: Theory and Implementation*, Ph.D. Thesis, University of Edinburgh (1995).

[22] V. Vasconcelos, *Typed concurrent objects* In Proc. ECOOP'94, Springer Lect. Notes in Comp. Sci. 821 (1994) 100-117.

Appendix: Substitutions

We recall that a name substitution is a mapping S from a finite subset $\mathsf{dom}(\mathsf{S})$ of $\mathcal{N}$ into $\mathcal{N}$, and that the result of applying S to a term U is denoted $[\mathsf{S}]U$, which is only defined if $\mathsf{im}(\mathsf{S}) \cap \mathsf{bn}(U) = \emptyset$. Since we only deal with capture-free substitutions, $[\mathsf{S}]U$ is trivially defined; let us just give the cases of the binding constructs, where by convention $\mathsf{S}(a) = a$ if $a \notin \mathsf{dom}(\mathsf{S})$:

$$
\begin{array}{rcll}
[\mathsf{S}]a(\vec{u}).P & = & \mathsf{S}(a)(\vec{u}).[\mathsf{S}']P & \mathsf{S}' = \mathsf{S}\restriction(\mathsf{dom}(\mathsf{S}) - \mathsf{nm}(\vec{u})) \\
[\mathsf{S}](\nu w)P & = & \left\{ \begin{array}{ll} (\nu a)[\mathsf{S}']P & \text{if } w = a \\ (\nu a@\mathsf{S}(\ell))[\mathsf{S}']P & \text{if } w = a@\ell \\ (\nu a : val)[\mathsf{S}']P & \text{if } w = a : val \end{array} \right\} & \text{where } \mathsf{S}' = \mathsf{S}\restriction(\mathsf{dom}(\mathsf{S}) - \{a\}) \\
[\mathsf{S}](\mathsf{rec}\, A(\vec{u}).P) & = & (\mathsf{rec}\, A(\vec{u}).[\mathsf{S}']P) & \mathsf{S}' = \mathsf{S}\restriction(\mathsf{dom}(\mathsf{S}) - \mathsf{nm}(\vec{u})) \\
[\mathsf{S}](\nu \ell : \psi)S & = & (\nu \ell : \mathsf{S}(\psi))[\mathsf{S}']S & \mathsf{S}' = \mathsf{S}\restriction(\mathsf{dom}(\mathsf{S}) - \{\ell\})
\end{array}
$$

where $\mathsf{S}(\psi)$ is defined in the obvious way. Similarly, the substitution of parametric processes for identifiers $[T/A]P$, with $\mathsf{fn}(T) \cap \mathsf{bn}(P) = \emptyset$, is essentially given by

$$
[T/A](\mathsf{rec}\, B(\vec{u}).P) = \left\{ \begin{array}{ll} (\mathsf{rec}\, B(\vec{u}).[T/A]P) & \text{if } A \neq B \\ (\mathsf{rec}\, B(\vec{u}).P) & \text{otherwise} \end{array} \right.
$$

The relation $=_\alpha$ of *α-conversion* is the least congruence satisfying the following axioms:

$$
\begin{array}{rcll}
a(\vec{u}).P & =_\alpha & a(\vec{u'}).[\vec{u'}/\vec{u}]P & \mathsf{nm}(\vec{u'}) \cap (\mathsf{nm}(P) \cup \{a\}) = \emptyset \\
(\nu a)U & =_\alpha & (\nu a')[a'/a]U & a' \notin \mathsf{nm}(U) \\
(\nu a@\ell)U & =_\alpha & (\nu a'@\ell)[a'/a]U & a' \notin \mathsf{nm}(U) \cup \{\ell\} \\
(\nu a : \tau)U & =_\alpha & (\nu a' : \tau)[a'/a]U & a' \notin \mathsf{nm}(U) \cup \mathsf{nm}(\tau) \\
(\mathsf{rec}\, A(\vec{u}).P) & =_\alpha & (\mathsf{rec}\, A'(\vec{u'}).[A'/A][\vec{u'}/\vec{u}]P) & A' \notin \mathsf{nm}(P),\ \mathsf{nm}(\vec{u'}) \cap \mathsf{nm}(P) = \emptyset
\end{array}
$$

Notice that, due to our implicit requirements, in the cases of $a(\vec{u}).P$ and $(\mathsf{rec}\, A(\vec{u}).P)$ the substitution $(\vec{u'}/\vec{u})$ has to be injective. The following result states that reduction is "compatible" with (term) substitution:

LEMMA A.1. *If $P \to P'$ then there exist Q and Q' such that $Q =_\alpha P$ and $Q' =_\alpha P'$, and $[T/A]Q \to [T/A]Q'$.*

PROOF: by induction on the inference of the reduction, straightforward. ▭

Now we establish the technical results needed to prove the subject reduction property. First we show that we can restrict the typing context to names that actually occur free in the typed term. In the following lemma, when we write $a \notin (\Gamma \vdash_\ell P)$ we mean that a does not occur in Γ (either in the domain or in the types assigned by this context), nor in P, and that $a \neq \ell$.

LEMMA A.2.

(i) *if $x : \tau, \Gamma \vdash_\ell P$ is provable and $x \notin \mathsf{fn}(P) \cup \{\ell\}$ then $\Gamma \vdash_\ell P$ is provable;*

(ii) *if $x : \tau, \Gamma \vdash_\ell T : \sigma$ is provable and $x \notin \mathsf{fn}(T) \cup \{\ell\}$ then $\Gamma \vdash_\ell T : \sigma$ is provable;*

(iii) *if $x : \tau, \Gamma \vdash S$ is provable and $x \notin \mathsf{fn}(S)$ then $\Gamma \vdash S$;*

(iv) *if $\ell : \{a : \gamma\}, \Gamma \vdash_\ell P$ is provable and $a \notin (\Gamma \vdash_\ell P)$ then $\Gamma \vdash_\ell P$ is provable;*

(v) *if $\ell : \{a : \gamma\}, \Gamma \vdash_\ell T : \sigma$ is provable and $a \notin (\Gamma \vdash_\ell T : \sigma)$ then $\Gamma \vdash_\ell T : \sigma$ is provable;*

(vi) *if $\ell : \{a : \gamma\}, \Gamma \vdash S$ is provable and $a \notin (\Gamma \vdash S)$ then $\Gamma \vdash S$ is provable.*

PROOF: by induction on the inference of the sequent, using the convention we made about the rules for input, restriction and recursion. Notice that if $x : \tau, \Gamma \vdash_\ell u : \sigma$ then $x \in \mathsf{nm}(u) \cup \{\ell\}$, and similarly if $\ell : \{a : \gamma\}, \Gamma \vdash_\ell u : \sigma$ then $a \in \mathsf{nm}(u) \cup \mathsf{nm}(\sigma)$. ▭

For the next lemma, we use a notion of *height* of a proof of a sequent, which should be clear – viewing a proof as a tree: an axiom has height 0, and each rule increases the maximum of the height of its premises by 1. In the statement of the lemma, we denote by $\mathsf{S}(\Gamma \vdash_\ell P)$ – and similarly for the other kinds of sequents – the sequent obtained by applying the substitution S to P, but also to the context, if this results in a

legal context, and to the current location; that is, $\mathsf{S}(\Gamma \vdash_\ell P) = \mathsf{S}(\Gamma) \vdash_{\mathsf{S}(\ell)} [\mathsf{S}]P$, where $\mathsf{S}(\Gamma)$ is defined in the obvious way. Similarly, when we consider sequents $\Gamma \vdash_\ell \overrightarrow{u : \tau}$ or $\Gamma \vdash_\ell A : Ch(\vec{\tau})$, we have to apply the substitution to the types $\vec{\tau}$, and we must be careful that this results in legal types.

LEMMA (SUBSTITUTION) A.3.

(i) *if* $\Gamma \vdash_\ell \overrightarrow{u : \tau}$ *is provable with a proof of heigth* h*, then so is* $\mathsf{S}(\Gamma \vdash_\ell \overrightarrow{u : \tau})$ *for any substitution* S *such that* $\mathsf{S}(\Gamma)$ *is a typing context and* S *is injective on* $\mathsf{nm}(\vec{\tau})$;

(ii) *if* $\Gamma \vdash_\ell P$ *(resp.* $\Gamma \vdash_\ell T : Ch(\vec{\tau})$*) is provable with a proof of heigth* h*, then so is* $\mathsf{S}(\Gamma \vdash_\ell P)$ *(resp.* $\mathsf{S}(\Gamma \vdash_\ell T : Ch(\vec{\tau}))$*) for any substitution* S *such that* $[\mathsf{S}]P$ *(resp.* $[\mathsf{S}]T$*) is defined and* $\mathsf{S}(\Gamma)$ *is a typing context (and* S *is injective on* $\mathsf{nm}(\vec{\tau})$*);*

(iii) *if* $\Gamma \vdash S$ *is provable with a proof of heigth* h*, then so is* $\mathsf{S}(\Gamma \vdash S)$*, for any substitution* S *such that* S *is injective on* $\mathsf{nm}(S)$*,* $[\mathsf{S}]S$ *is defined and* $\mathsf{S}(\Gamma)$ *is a typing context;*

(iv) *if* $\Delta \vdash_\ell T : Ch(\vec{\tau})$ *and* $A : Ch(\vec{\tau}), \Gamma \vdash_\ell P$ *(resp.* $A : Ch(\vec{\tau}), \Gamma \vdash_\ell T' : \sigma$*, resp.* $A : Ch(\vec{\tau}), \Gamma \vdash S$*) are provable, and* A *does not occur in* Γ*, then* $\Delta, \Gamma \vdash_\ell [T/A]P$ *(resp.* $\Delta, \Gamma \vdash_\ell [T/A]T' : \sigma$*, resp.* $\Delta, \Gamma \vdash [T/A]S$*) is provable, provided that no free name of* T *is bound in* P *(resp.* T'*,* S*).*

PROOF:

(i) by induction on h, easy.

(ii) by induction on h, and by case on the last rule used to infer the sequent. If P is $a(\vec{u}).R$ and $\Gamma = \ell : \{a : Ch(\vec{\tau})\}, \Delta$ with $\ell : \{a : Ch(\vec{\tau})\}, \Theta, \Delta \vdash_\ell R$ where $\Theta \vdash_\ell \overrightarrow{u : \tau}$, then $\mathsf{S}'(\ell : \{a : Ch(\vec{\tau})\}, \Theta, \Delta \vdash_\ell R)$ is provable, by induction hypothesis, where $\mathsf{S}' = \mathsf{S} \restriction \mathsf{dom}(\mathsf{S}) - \mathsf{nm}(\vec{u})$. Since $\mathsf{S}'(\Theta) \vdash_\ell \overrightarrow{u : \mathsf{S}'(\tau)}$, we may use the rule for typing the input construct to conclude. The case of $P = (\mathsf{rec}\, A(\vec{u}).Q)$ is similar. All the other cases are very easy. The proof of (iii) is similar.

(iv) by induction on the inference of $A : Ch(\vec{\tau}), \Gamma \vdash_\ell U$ or $A : Ch(\vec{\tau}), \Gamma \vdash_\ell T' : \sigma$. If this sequent is an axiom $A : Ch(\vec{\tau}) \vdash_\ell A : Ch(\vec{\tau})$ then the conclusion is the hypothesis $\Delta \vdash_\ell T : Ch(\vec{\tau})$. The other cases are equally easy. ❑

LEMMA A.4. *Let* S *be an injective substitution such that* $\Gamma \vdash_\ell [\mathsf{S}]P$ *(resp.* $\Gamma \vdash_\ell [\mathsf{S}]T : \sigma$*, resp.* $\Gamma \vdash [\mathsf{S}]S$*) is provable, with a proof of height* h*. Then so is* $\mathsf{S}^{-1}(\Gamma) \vdash_{\ell'} P$ *(resp.* $\mathsf{S}^{-1}(\Gamma) \vdash_{\ell'} T : [\mathsf{S}^{-1}]\sigma$*, resp.* $\mathsf{S}^{-1}(\Gamma) \vdash S$*) where* $\mathsf{S}(\ell') = \ell$.

PROOF: by induction on the inference of the sequent, and by case on the last rule used in this inference. Let us just examine the case where $[\mathsf{S}]P = a(\vec{u}).R$ and $\Gamma = \ell : \{a : Ch(\vec{\tau})\}, \Delta$ with $\Theta, \Delta \vdash_\ell R$ and $\Theta \vdash_\ell \vec{u} : \vec{\tau}$. We have $P = a'(\vec{u}).R'$ with $a = \mathsf{S}(a')$ and $R = [\mathsf{S}_0]R'$ where $\mathsf{S}_0 = \mathsf{S} \restriction \mathsf{dom}(\mathsf{S}) - \mathsf{nm}(\vec{u})$. Since we are assuming that $[\mathsf{S}]P$ is defined, we have $\mathsf{im}(\mathsf{S}) \cap \mathsf{nm}(\vec{u}) = \emptyset$. By induction hypothesis $\mathsf{S}_0^{-1}(\Theta, \Delta) \vdash_{\ell'} R'$, and $\mathsf{S}_0^{-1}(\Theta) \vdash_{\ell'} \overrightarrow{u : [\mathsf{S}_0^{-1}]\tau}$ by the Substitution Lemma A.3(i). Since the names of $\mathsf{nm}(\vec{u})$ do not occur in $\ell : \{a : Ch(\vec{\tau})\}, \Delta$, we have $[\mathsf{S}_0^{-1}]\vec{\tau} = [\mathsf{S}^{-1}]\vec{\tau}$ and $\mathsf{S}_0^{-1}(\Delta) = \mathsf{S}^{-1}(\Delta)$. Let $\vec{\theta} = [\mathsf{S}^{-1}]\vec{\tau}$. A name $b \in \mathsf{nm}(\vec{u})$ cannot occur in $\ell' : \{a' : Ch(\vec{\theta})\}, \mathsf{S}^{-1}(\Delta)$, since otherwise we would have $b \in \mathsf{im}(\mathsf{S})$, or b would occur in Γ. Therefore we may conclude $\mathsf{S}^{-1}(\Gamma) \vdash_{\ell'} P$. The case of $\Gamma \vdash_\ell [\mathsf{S}](\mathsf{rec}\, A(\vec{u}).R) : \sigma$ is similar, and the other ones are easy. ❑

Now we prove that typing is preserved by α-conversion.

LEMMA A.5. *The relation* $=_{\mathcal{T}}$ *contains the relation* $=_\alpha$ *of* α*-conversion.*

PROOF: by induction on the definition of $=_\alpha$ (since $=_{\mathcal{T}}$ is a congruence we only have to consider the axioms of α-conversion). We show that if $\Gamma \vdash_\ell a(\vec{u}).P$ and $(\vec{v}/\vec{u})$ is an injective substitution such that $\mathsf{nm}(\vec{v}) \cap (\mathsf{nm}(P) \cup \{a\}) = \emptyset$, then $\Gamma \vdash_\ell a(\vec{v}).[\vec{v}/\vec{u}]P$, by induction on the height of the inference of $\Gamma \vdash_\ell a(\vec{u}).P$. Let S be an injective substitution such that $\mathsf{dom}(\mathsf{S}) = \mathsf{nm}(\vec{v})$ and the names in $\mathsf{im}(\mathsf{S})$ are fresh. Then $\mathsf{S}(\Gamma) \vdash_{\ell'} a(\vec{u}).P$, where $\ell' = \mathsf{S}(\ell)$, with a proof of the same height as the one of $\Gamma \vdash_\ell a(\vec{u}).P$, by the Substitution Lemma A.3(ii). There are two cases: if $\mathsf{S}(\Gamma) = \ell : \{a : Ch(\vec{\tau})\}, \Delta$ with $\Theta, \Delta \vdash_{\ell'} P$ and $\Theta \vdash_{\ell'} \overrightarrow{u : \tau}$, and the names of $\mathsf{nm}(\vec{u})$ do not occur in $\mathsf{S}(\Gamma)$ (and are distinct from ℓ'), then $(\vec{v}/\vec{u})(\Theta), \Delta \vdash_{\ell'} [\vec{v}/\vec{u}]P$ by the Substitution Lemma A.3(ii). and $(\vec{v}/\vec{u})(\Theta) \vdash_{\ell'} \overrightarrow{v : \tau}$ by the Substitution Lemma A.3. (the names of $\vec{u}$ cannot occur in $\vec{\tau}$, since they do not occur in $\mathsf{S}(\Gamma)$). Since we have chosen S such that the names of $\mathsf{nm}(\vec{v})$ do not occur in $\mathsf{S}(\Gamma)$ (and are distinct from ℓ'), we may infer $\mathsf{S}(\Gamma) \vdash_{\ell'} a(\vec{v}).[\vec{v}/\vec{u}]P$, and therefore $\Gamma \vdash_\ell a(\vec{v}).[\vec{v}/\vec{u}]P$ by the Lemma A.4, since $a(\vec{v}).[\vec{v}/\vec{u}]P = [\mathsf{S}]a(\vec{v}).[\vec{v}/\vec{u}]P$. Otherwise, if $\mathsf{S}(\Gamma) = \Delta', \Delta$ with $\Delta \vdash_{\ell'} a(\vec{u}).P$, we use the induction hypothesis, and conclude in a similar way.

To prove the symmetric case, that is $\Gamma \vdash_\ell a(\vec{v}).[\vec{v}/\vec{u}]P$ implies $\Gamma \vdash_\ell a(\vec{u}).P$, we proceed in a similar way, first renamming the names of $\vec{u}$ (which do not include a) in Γ with fresh names, and using the Lemma A.4. The other cases are similar, or even easier. ❑

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399